Towards Reversible Sessions[*]

Francesco Tiezzi
IMT Institute for Advanced Studies, Lucca, Italy
francesco.tiezzi@imtlucca.it

Nobuko Yoshida
Imperial College, London, U.K.
n.yoshida@imperial.ac.uk

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 taking different computation paths along the same session, as well as reverting the whole session and starting 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 enrich a session-based variant of π-calculus with memory devices, dedicated to keep track of the computation history of sessions in order to reverse it. We discuss our initial investigation concerning the definition of a session type discipline for the proposed reversible calculus, and its practical advantages for static verification of safe composition in communication-centric distributed software performing reversible computations.

1 Introduction

Reversible computing aims at providing a computational model that, besides the standard (forward) executions, also permits backward execution steps in order to undo the effect of previously performed forward computations. Reversibility is a key ingredient in different application domains since many years and, recently, also in the design of reliable concurrent systems, as it permits understanding existing patterns for programming reliable systems (e.g., compensations, checkpointing, transactions) and, possibly, improving them or developing new ones.

A promising line of research on this topic advocates reversible variants of well-established process calculi, such as CCS and π-calculus, as formalisms for studying reversibility mechanisms in concurrent systems. Our work incorporates reversibility into a variant of π-calculus equipped with *session* primitives supporting structured 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 static checking framework to guarantee the correctness of communication 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,

[*]This work has been partially supported by the COST Action BETTY (IC1201), by the EU project ASCENS (257414), and by the Italian MIUR PRIN project CINA (2010LHT4KM).

etc.). Then, the client can either accept, negotiate or reject the quote. If a problem occurs during the interaction between the client and the provider, the computation can be reverted, in order to allow the client to automatically start a new session with (possibly) another provider.

The proposed reversible session-based 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 each 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, i.e. 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.

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. On the other hand, the session type discipline affects reversibility as it forces concurrent interactions to follow structured communication patterns. In fact, linearizing behaviours on sessions reduces the effect of causal consistency, because concurrent interactions along the same session are forbidden and, hence, the rollback along a session follows a single path. However, interactions along different sessions are still concurrent and, therefore, they can be reverted as usual 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, irreversible closure of sessions).

2 Related work

We review here the most closely related works, which concern the definition of reversible process calculi; we refer to [9] for a more detailed review of reversible calculi.

Reversible CCS (RCCS) [3] is the first proposal of reversible calculus, from which all subsequent works drew inspiration. 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 roll-back. 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 indicating which of the two sons the thread is. A drawback of this approach is that the parallel operator does not satisfy usual structural congruence rules as commutativity, associativity and nil process as neutral element.

CCS-R [4] is another reversible variant of CCS, which mainly aims at formalising biological systems. Like RCCS, it relies on memory stacks for storing data needed for backtracking, which now also includes events corresponding to unfolding of process definitions. Differently from RCCS, specific identifiers are used to label threads, and a different approach is used for dealing with forking.

CCS with communication Keys (CCSK) [8] 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. To properly revert synchronisations, the two threads have to agree on a key, uniquely identifying that communication.

$\rho\pi$ [7] is a reversible variant of the higher-order π-calculus. It borrows from RCCS the use of memories for keeping track of past actions, although in $\rho\pi$ they are not stacks syntactically associated to threads, but parallel terms each one dedicated to a single communication. The connection between memories and threads is kept by resorting to identifiers, which resemble CCSK keys. Fork handling is based on structured tags, used to connect the identifier of a thread with the identifiers of its sub-threads. This approach to reversibility has been applied in [5] to a distributed tuple-based language.

Another reversible variant of π-calculus is Rπ [2]. Similarly to RCCS, this calculus relies on memory stacks, now recording communication events and forking. Differently from $\rho\pi$, it considers standard π-calculus (without choice and replication) as a host calculus and its semantics is defined in terms of a labelled transition relation.

Finally, reversible structures [1] is a simple computational calculus for modelling chemical systems. Reversible structures does not exploit memories, but maintains the structure of terms and uses a special symbol to indicate the next forward and backward operations that a term can perform.

In our work, we mainly take inspiration from the $\rho\pi$ approach. In fact, all other approaches are based on CCS and 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 extrusion.

3 Reversible Session-based π-calculus

In this section, we introduce a reversible extension of a π-calculus enriched with primitives for managing *binary* sessions, i.e. structured interactions between two parties. We call *ReSπ* (*Reversible Session-based* π-calculus) this formalism. Due to lack of space, some technical details about semantics and results have been omitted; we refer the interested reader to [9] for a more complete account.

From π-calculus to ***ReSπ*****.** Our approach to keep track of computation history in *ReSπ* is as follows: we tag processes with unique identifiers (tagged processes are called *threads*) 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 computations* (denoted by $\twoheadrightarrow$), also *backward computations* (denoted by $\rightsquigarrow$) that undo the effect of the former ones in a causal-consistent fashion.

π-calculus *processes* and *expressions* are given by the grammars in Figure 1. The synchronisation on a shared channel a of processes $\bar{a}(x).P$ and $a(y).Q$ initiates a session along a fresh session channel s. This channel consists in a pair of (dual) endpoints, denoted by s and $\bar{s}$ (such that $\bar{\bar{s}} = s$), each one dedicated to one party to exchange values with the other. These 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 realise the standard synchronous message passing, where messages result from expressions evaluation and may contain endpoints (*delegation*). Constructs $k \triangleleft l.P$ and $k' \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$ denote label selection and branching (with l_1, ..., l_n pairwise distinct) via k and k', respectively. The above interaction primitives are combined by conditional choice, parallel composition, restriction, recursion and inaction.

Shared channels $a, b, \ldots$	Session channels $s, s', \ldots$	Session endpoints $s, \bar{s}, s', \bar{s}', \ldots$	Variables $x, y, \ldots$
Labels $l, l', \ldots$	Process variables $X, Y, \ldots$	Tags $t, t', \ldots$	Shared ids $u ::= a \mid x$
Channels $c ::= a \mid s$	Names $h ::= c \mid t$	Session ids $k ::= s \mid \bar{s} \mid x$	

$$\begin{array}{rrcl}
\textbf{Processes} & P & ::= & \bar{u}(x).P \mid u(x).P \mid k!\langle e\rangle.P \mid k?(x).P \mid k \triangleleft l.P \mid k \triangleright \{l_1 : P_1, \ldots, l_n : P_n\} \\
& & \mid & \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \mid P \mid Q \mid (\nu c)P \mid X \mid \mu X.P \mid \mathbf{0} \\
\textbf{Expressions} & e & ::= & v \mid x \mid \mathrm{op}(e_1, \ldots, e_n) \\
\textbf{Values} & v & ::= & \texttt{true} \mid \texttt{false} \mid 0,1,\ldots \mid a \mid s \mid \bar{s} \\
\textbf{\textit{ReS}}\boldsymbol{\pi}\ \textbf{processes} & M & ::= & t : P \mid (\nu h)M \mid M \mid N \mid m \mid \mathrm{nil} \\
\textbf{Memories} & m & ::= & \langle t_1 \twoheadrightarrow_{\mathrm{A}} t_2, t_1', t_2'\rangle \mid \langle t, e?P:Q, t'\rangle \mid \langle t \rightrightarrows (t_1, t_2)\rangle \\
& \mathrm{A} & ::= & a(x)(y)(\nu s)PQ \mid k\langle e\rangle(x)PQ \mid k \triangleleft l_i P\{l_1 : P_1, \ldots, l_n : P_n\}
\end{array}$$

Figure 1: *ReSπ* syntax

ReSπ processes are built upon π-calculus processes by labelling them with *tags* to uniquely identify threads $t : P$. Uniqueness of tags is ensured by using the restriction operator and by only considering *reachable* terms (Def. 1). Moreover, *ReSπ* extends π-calculus with three kinds of *memories m*. An *action memory* $\langle t_1 \twoheadrightarrow_{\mathrm{A}} t_2, t_1', t_2'\rangle$ stores an action event A together with the tag t_1 of the active party of the action, the tag t_2 of the passive party, and the tags t_1' and t_2' of the new threads activated by the corresponding reduction. An *action event* records information necessary to revert each kind of interactions, which can be either a session initiation $a(x)(y)(\nu s)PQ$, a communication along an established session $k\langle e\rangle(x)PQ$, or a branch selection $k \triangleleft l_i P\{l_1 : P_1, \ldots, l_n : P_n\}$. A *choice memory* $\langle t, e?P:Q, t'\rangle$ stores a choice event together with the tag t of the conditional choice and t' of the new activated thread. The event $e?P:Q$ records the evaluated expression e, and processes P and Q of the then- and else-branch, respectively. A *fork memory* $\langle t \rightrightarrows (t_1, t_2)\rangle$ stores the tag t of a splitting thread, of the form $t : (P \mid Q)$, and the tags t_1 and t_2 of the new activated threads $t_1 : P$ and $t_2 : Q$; these memories are analogous to *connectors* in [5]. Threads and memories are composed by parallel composition and restriction operators.

Not all processes allowed by the syntax are semantically meaningful. In a general term, 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 $\langle t' \twoheadrightarrow_{\mathrm{A}} t_2, t_1', t_2'\rangle$, $\langle t_1 \twoheadrightarrow_{\mathrm{A}} t', t_1', t_2'\rangle$, $\langle t', e?P_1:P_2, t_1\rangle$, and $\langle t' \rightrightarrows (t_1, t_2)\rangle$ exists. Thus, as in [2], to ensure history consistency we only consider *reachable* processes, i.e. processes obtained by means of forward and backward reductions from processes with unique tags and no memory.

Def. 1 (Reachable processes) *The set of* reachable *ReSπ processes is the closure under $\rightarrowtail$ (see below) of the set of terms, whose threads have distinct tags, generated by* $M ::= t : P \mid (\nu c)M \mid M \mid N \mid \mathrm{nil}$.

***ReSπ* semantics.** The *ReSπ* operational semantics is given in terms of a *reduction relation* $\rightarrowtail$, given as the union of the forward and backward reduction relations. We report here, by way of examples, the forward and backward rules for session initiation (we require $s, \bar{s}$ fresh in P_1 and P_2 in the forward rule):

$$t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2 \twoheadrightarrow (\nu s, t_1', t_2')(t_1' : P_1[\bar{s}/x] \mid t_2' : P_2[s/y] \mid \langle t_1 \twoheadrightarrow_{a(x)(y)(\nu s)P_1P_2} t_2, t_1', t_2'\rangle)$$

$$(\nu s, t_1', t_2')(t_1' : P \mid t_2' : Q \mid \langle t_1 \twoheadrightarrow_{a(x)(y)(\nu s)P_1P_2} t_2, t_1', t_2'\rangle) \rightsquigarrow t_1 : \bar{a}(x).P_1 \mid t_2 : a(y).P_2$$

When two parallel threads synchronise to establish a new session, two fresh tags are created to uniquely identify the continuations. Moreover, all relevant information is stored in the action memory: 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 in the backward rule to revert this reduction. 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). When considering reachable processes, due to tag uniqueness, processes P and Q 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. Therefore, the fact that two threads tagged with t_1' and t_2' are in parallel with 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 undone the forward computation itself.

Multiple providers scenario. The scenario involving a client and two providers informally introduced in Section 1 is rendered in *ReSπ* as $(t_1 : P_{client} \mid t_2 : P_{provider1} \mid t_3 : P_{provider2})$, where the client process P_{client} is

$$\overline{a_{login}}(x).x!\langle \mathsf{srv\_req}\rangle.x?(y_{quote}).\mathtt{if}\ accept(y_{quote})\ \mathtt{then}\ x \triangleleft l_{acc}.P_{acc}$$
$$\mathtt{else}\ (\mathtt{if}\ negotiate(y_{quote})\ \mathtt{then}\ x \triangleleft l_{neg}.P_{neg}\ \mathtt{else}\ x \triangleleft l_{rej}.\mathbf{0})$$

while $P_{provider\,i}$ is

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

If the client contacts the first provider and accepts the proposed quote, the system evolves to

$$M = (\nu s,\ldots,t_1',t_2')(t_1' : P_{acc}[\bar{s}/x, \mathsf{quote}/y_{quote}] \mid t_2' : Q_{acc}[s/y, \mathsf{srv\_req}/z_{req}] \mid m_1 \mid \ldots \mid m_5) \mid P_{provider2}$$

where memories m_i keep track of the computation history. Now, if a problem occurs during the subsequent interactions, the computation can be reverted to allow the client to start a new session with (possibly) another provider:

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

Properties of *ReSπ*. We show here that *ReSπ* enjoys standard properties of reversible calculi.

First, we demonstrate 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. This decoration can be erased by means of the *forgetful map* ϕ, mapping *ReSπ* terms into π-calculus ones by removing memories, tag annotations and tag restrictions. The following lemmas show that each forward reduction of a *ReSπ* process corresponds to a reduction of the corresponding π-calculus process and vice versa.

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

Lemma 2 *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 $M \twoheadrightarrow N$.*

Then, we show that *ReSπ* backward reductions are the inverse of the forward ones and vice versa.

Lemma 3 (Loop lemma) *Let M and N be two reachable ReSπ processes. $M \twoheadrightarrow N$ if and only if $N \rightsquigarrow M$.*

We conclude with the *causal consistency* result stating that two sequences of reductions (called *traces* and ranged over by σ), with the same initial state (*coinitial*) and equivalent w.r.t. the standard notion of *causal equivalence* ($\asymp$), lead to the same final state (*cofinal*). Thus, in this case, we can rollback to the initial state by reversing any of the two traces.

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.*

4 Discussion on a type discipline

A question that should be answered before defining a static type discipline for a reversible calculus is "*Should we type check the processes stored in the memories?*". The question arises from the fact that we should be able to determine if any *ReSπ* process is well-typed or not. In our case the answer is "*Yes*", otherwise typability would not be preserved under reduction (i.e., Subject Reduction would not be satisfied). It is indeed easy to define a *ReSπ* process (see [9]) containing a memory that, even if consistent, triggers a backward reduction leading to an untypable term (by the type system defined in [10] for the host calculus).

One could wonder now if it is possible to type *ReSπ* processes in a naïve way by separately type checking the term resulting from the application of ϕ and each single memory, by using the type system in [10]. For each memory we would check the term that has triggered the forward reduction generating the memory. In general, this approach does not work, because the term stored in a memory cannot be type checked in isolation without taking into account its context. For example, consider a memory corresponding to a communication along a session s typable under typing $\Delta = \bar{s} : ![\mathsf{int}].\mathsf{end} \cdot s : ?[\mathsf{int}].\mathsf{end}$ and in parallel with $(t_1 : \bar{s}!\langle 1 \rangle \mid t_2 : s?(x))$. The term resulting from the corresponding backward reduction is not typable, because the typings of its sub-terms are not composable (indeed, $\Delta \cdot \Delta$ is not defined).

Memory context can be considered by extending the type system in [10] with rules that permits typing (processes stored in) memories and ignoring tag annotations and tag restrictions (see [9] for the definition of this type system). In this way, during type checking, typings of memories and threads must be composed by means of the rule for parallel composition. Thus, e.g., the *ReSπ* process mentioned above is, rightly, untypable. This type system properly works only on a simplified setting, which permits avoiding to deal with dependencies among memories and the threads outside memories, that could cause unwanted conflicts during type checking. Specifically, we consider the class of *ReSπ* processes that, extending Def. 1, are obtained by means of forward and backward reductions from processes with unique tags, no memory, no session initialised, no conditional choices and recursions at top-level, and no delegation. The characteristic of these processes is that, for each memory inside a process, there exists within the process an ancestor memory corresponding to the initialisation of the considered session. The type system checks only this latter kind of memories, which significantly simplifies the theory.

Coming back now to the multiple providers scenario, we can verify that the initial process is well-typed. 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 sorts Request and Quote are used to type requests and quotes, respectively. Let us consider now a scenario where the client wills to concurrently submit two different requests to the same provider, which would concurrently serve them. Consider in particular the following specification of the client

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

The new specification is clearly not well-typed, due to the use of parallel threads within the same session. This permits avoiding 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.

5 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.

The type discipline for $ReS\pi$ is still subject of study. In fact, the type system mentioned in Section 4 is not completely satisfactory, because its use is limited to a restricted class of processes. To consider a broader class, an appropriate static type checking approach for memories has to be devised. For each memory, we would check a term composed of the threads stored in the memory and of a context composed of threads that have not been generated by the execution of the memory threads.

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 [7]). It is worth noticing that the 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. In line with [6], we plan to enrich the language with mechanisms to control reversibility. Moreover, we intend to enrich the framework with an *irreversible* action for committing the closure of sessions. In this way, computation would go backward and forward, allowing the parties to try different interactions, until the session is successfully completed. For instance, the process P_{acc} in our example could terminate by performing the irreversible action $\texttt{commit}(x)$, which has to synchronise with action $\texttt{commit}(y)$ in Q_{acc}. Differently from sessions terminated by $\mathbf{0}$, a session terminated by a `commit` synchronisation is unbacktrackable. The irreversibility is due to the fact that no backward rule is defined to revert this interaction. The type theory should be tailored to properly deal with this kind of session closure.

As longer-term goals, we intend to apply the proposed approach to other session-based formalisms, which consider, e.g., asynchronous sessions and multiparty sessions. Moreover, we plan to investigate implementation issues that may arise when incorporating the approach into standard programming languages, in particular in case of a distributed setting.

References

[1] Luca Cardelli & Cosimo Laneve (2011): *Reversible structures*. In: *CMSB*, ACM, pp. 131–140, doi:10.1145/2037509.2037529.

[2] I. Cristescu, J. Krivine & D. Varacca (2013): *A Compositional Semantics for the Reversible p-Calculus*. In: *LICS*, IEEE, pp. 388–397, doi:10.1109/LICS.2013.45.

[3] Vincent Danos & Jean Krivine (2004): *Reversible Communicating Systems*. In: *CONCUR*, *LNCS* 3170, Springer, pp. 292–307, doi:10.1007/978-3-540-28644-8_19.

[4] Vincent Danos & Jean Krivine (2007): *Formal Molecular Biology Done in CCS-R*. *Electr. Notes Theor. Comput. Sci.* 180(3), pp. 31–49, doi:10.1016/j.entcs.2004.01.040.

[5] E. Giachino, I. Lanese, C.A. Mezzina & F. Tiezzi (2013): *Causal-Consistent Reversibility in a Tuple-Based Language*. Technical Report. `http://www.cs.unibo.it/~lanese/work/klaimrev-TR.pdf`.

[6] I. Lanese, C.A. Mezzina, A. Schmitt & J. Stefani (2011): *Controlling Reversibility in Higher-Order Pi*. In: *CONCUR*, *LNCS* 6901, Springer, pp. 297–311, doi:10.1007/978-3-642-23217-6_20.

[7] I. Lanese, C.A. Mezzina & J. Stefani (2010): *Reversing Higher-Order Pi*. In: *CONCUR*, *LNCS* 6269, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[8] Iain C. C. Phillips & Irek Ulidowski (2007): *Reversing algebraic process calculi*. *J. Log. Algebr. Program.* 73(1-2), pp. 70–96, doi:10.1016/j.jlap.2006.11.002.

[9] Francesco Tiezzi & Nobuko Yoshida (2014): *Towards Reversible Sessions*. Technical Report. `http://cse.lab.imtlucca.it/~tiezzi/papers/places2014_full.pdf`.

[10] N. Yoshida & V.T. Vasconcelos (2007): *Language Primitives and Type Discipline for Structured Communication-Based Programming Revisited: Two Systems for Higher-Order Session Communication*. *Electr. Notes Theor. Comput. Sci.* 171(4), pp. 73–93, doi:10.1016/j.entcs.2007.02.056.