Reversible Sessions Using Monitors*

Claudio A. Mezzina
IMT School for Advanced Studies Lucca, Italy

Jorge A. Pérez
University of Groningen, The Netherlands

Much research has studied foundations for correct and reliable *communication-centric systems*. A salient approach to correctness uses *session types* to enforce structured communications; a recent approach to reliability uses *reversible* actions as a way of reacting to unanticipated events or failures. This note develops a simple observation: the machinery required to define asynchronous semantics and monitoring can also support reversible protocols. We propose a process framework of session communication in which monitors support reversibility. A key novelty in our approach are *session types with present and past*, which allow us to streamline the semantics of reversible actions.

1 Introduction

Much research has studied foundations for reliable *communication-centric* software systems. Our interest is in process frameworks that, building on core calculi for concurrency, offer analysis techniques for message-passing programs. While early frameworks focused on (static) verification of protocol correctness, as enforced by properties such as safety, fidelity, and progress (deadlock-freedom) (see, e.g., [15, 16, 4]), extensions with external mechanisms (such as, e.g., exceptions, interruptions, and compensations [3, 12, 6], adaptation [14], monitoring [18], reversibility [25]) have been proposed to enforce protocol correctness even in the presence of unanticipated events (say, failures or new requirements).

Comprehensive approaches to correctness and reliability, which address and enforce both kinds of requirements, seem indispensable in the principled design of communication-centric systems. As these systems are increasingly built using heterogeneous services whose provenance/correctness cannot always be certified in advance, static validation techniques (such as type systems) fall short. Correctness must then be guaranteed by mechanisms for reliability, which may inspect the (visible) behavior of interacting services and take action if they deviate from prescribed communication protocols.

We report on ongoing work aimed at uniform approaches to correct, reliable communicating systems. We address the interplay between *session types* and models of *reversible computation*: models of concurrency in which the usual *forward* semantics is coupled with a *backward* semantics that allows one to "undo" process actions [11]. We explore to what extent type information can streamline the reversible semantics for interacting processes. Our discovery is that known (run-time) mechanisms used to support asynchronous (queue-based) semantics and monitoring can also support reversible protocol actions.

A key technical device in formalizing reversible semantics are *memories*: these are run-time constructs which enable one to revert actions. Memories are the bulk of a reversible model; their maintenance requires care, as demonstrated by Tiezzi and Yoshida [25], who adapt known reversible semantics [11, 23] into the session typed setting. In this work, we explore a different approach: we use monitors as memories. We investigate to what extent queue-based semantics for session types can support reversibility. The key idea is simple: we use the type-checking component of queue-based semantics (i.e., the fact that session types enable process reductions) to support reversible process actions. Our approach concerns directly the reduction semantics for sessions, so we illustrate it via approximate reduction rules,

*Research partly supported by EU COST Actions IC1201 and IC1405.

D. Orchard and N. Yoshida (Eds.): Programming Language Approaches to Concurrency- and Communication-Centric Software (PLACES 2016).
EPTCS 211, 2016, pp. 56–64, doi:10.4204/EPTCS.211.6

which omit unimportant notational details. Consider the reduction rule for session communication, enhanced with session types and message queues, in the style of [19, 17, 21]:

$$\overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor !U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor ?U.S_2 \cdot \tilde{h}_2\rfloor \longrightarrow P \parallel \overline{s}\lfloor S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor S_2 \cdot \tilde{h}_2, v\rfloor \tag{1}$$

In (1), processes $\overline{s}\langle v\rangle.P$ and $s(x).Q$ denote output and input along *session endpoints* $\overline{s}$ and s, respectively. Notice that $\overline{s}$ and s are *dual* endpoints. Given an endpoint s, process $s\lfloor S \cdot \tilde{h}\rfloor$ is a *monitor*, where S and $\tilde{h}$ are the session type and message queue for s, respectively. In the approach of [19, 17, 21], session types enable communication actions: a synchronization can only occur if the actions (in the processes) correspond to the intended protocols (in the monitor types). After synchronization, portions of both processes and monitor types are consumed. Our approach consists in keeping, rather than consuming, these monitor types. For this to work, we need to distinguish the part of the protocol that has been already executed (its past), from the protocol that still needs to execute (its present). We thus introduce session types with *present and past*: the type $S\hat{}T$ says that actions abstracted by S are past protocol actions, whereas actions in T are present steps. We may refine (1) as follows:

$$\overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor T\hat{}!U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor T'\hat{}?U.S_2 \cdot \tilde{h}_2\rfloor \twoheadrightarrow P \parallel \overline{s}\lfloor T.!U\hat{}S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor T'.?U\hat{}S_2 \cdot \tilde{h}_2, v\rfloor \tag{2}$$

This is a *forward* reduction rule. Monitors $\overline{s}\lfloor T\hat{}!U.S_1 \cdot \tilde{h}_1\rfloor$ and $s\lfloor T'\hat{}?U.S_2 \cdot \tilde{h}_2\rfloor$ use type-checking to enable forward and backward computations; they may also implement asynchronous communication. Observe that we use the cursor $\hat{}$ to preserve output and input protocol actions (noted $!U$ and $?U$, respectively). Based on (2), we may state a corresponding *backward* reduction rule, which reverts the intra-session synchronization at the level of processes, types, and message queues:

$$P \parallel \overline{s}\lfloor T_1.!U\hat{}S_1 \cdot \tilde{h}_1\rfloor \parallel Q \parallel s\lfloor T_2.?U\hat{}S_2 \cdot \tilde{h}_2, v\rfloor \rightsquigarrow \overline{s}\langle v\rangle.P \parallel \overline{s}\lfloor T_1\hat{}!U.S_1 \cdot \tilde{h}_1\rfloor \parallel s(x).Q \parallel s\lfloor T_2\hat{}?U.S_2 \cdot \tilde{h}_2\rfloor \tag{3}$$

Our main technical contribution is a core framework for session communication and reversibility whose monitored semantics follows the spirit of rules (2) and (3). In our framework, session processes occur within *configurations*, which add monitors and state for endpoints: while state conveniently implements substitutions, monitors handle both communication and reversibility. Reduction is defined for configurations following rules (2) and (3). We support session establishment and the consistent use of sent values and open variables in the state (cf. v and x in (2) and (3)). Our semantics enjoys the so-called "loop lemma", which offers a basic consistency guarantee for the interplay of forward and backward actions.

In our opinion, the use of monitors with type-checking for reversible semantics is an observation that has at least two significant implications. First, it is encouraging to discover that monitor-based semantics with type-checking—introduced in [19, 17, 21] for asynchronous communications with events and used in [13, 9] to define run-time adaptation—may also inform the semantics of reversible protocols. Monitors have also been used for security purposes [5, 8] and, quite recently, for assigning blame to deviant session processes [18]. Therefore, a monitor-based semantics encompasses an array of seemingly distinct concerns in structured communications. Second, we see our developments as a first step towards validation techniques for communication and reversibility based on run-time verification. Session frameworks with run-time verification have been developed in, e.g., [2, 12]. As these works do not support reversibility, our work may suggest enhancements for their dynamic verification capabilities.

2 Syntax and Semantics

In this section we present our framework of session processes with monitored semantics and reversibility. We assume the following denumerable infinite mutually disjoint sets: the set $\mathscr{S}$ of *session names* (or

$$
\begin{array}{rcl}
k,k' ::= & s,\overline{s} \mid x,y & \qquad u,u' ::= \quad a,b \mid x,y \qquad\qquad n,m ::= \quad a,b \mid s,\overline{s} \\
M,N ::= & \mathbf{0} \mid \langle P,\sigma\rangle_{\delta} \mid a\lfloor H\cdot\widetilde{x}\cdot\widetilde{u}\rfloor \mid \nu n.M \mid M \parallel N & \\
P,Q ::= & u(x:S).P \mid \overline{u}(x:S).P \mid \overline{k}\langle v\rangle.P \mid k(x).P \mid \nu a.P \mid \mathbf{0} & \\
S,T ::= & \mathtt{end} \mid !U.S \mid ?U.S & \\
H,K ::= & \hat{}S \mid S\hat{} \mid S\hat{}T &
\end{array}
$$

Figure 1: Syntax of Configurations, Processes, and Session Types.

endpoints), the set $\mathscr{C}$ of *channels* and the set of *variables* $\mathscr{X}$. The set $\mathscr{N} = \mathscr{S} \cup \mathscr{C}$ is called the set of *names*. We assume a total bijection over $\mathscr{N}$, noted $\overline{\cdot}$, relating names with their duals such that $\overline{n} \neq n$ and $\overline{\overline{n}} = n$, for any name n. We let a,b to range over $\mathscr{C}$; s,r (and their duals) to range over $\mathscr{S}$; m,n to range over $\mathscr{N}$ and x,y to range over $\mathscr{X}$. We use $\widetilde{o}$ to denote a finite sequence of objects (names, sessions, variables) $o_1,o_2,\ldots,o_n$, which we sometimes treat as a set or as an ordered list. We write δ, δ' to range over finite, possibly empty sequences of session names.

Syntax. The syntax of configurations M,N, processes P,Q, and session types S,T is given in Fig. 1. The syntax of M includes the empty configuration $\mathbf{0}$, the *running* process $\langle P,\sigma\rangle_{\delta}$, a monitor $s\lfloor S\cdot\widetilde{x}\cdot\widetilde{u}\rfloor$, the name restriction $\nu n.M$, and parallel composition $M \parallel N$. A running process $\langle P,\sigma\rangle_{\delta}$ is univocally identified by δ, the sequence of session endpoints occurring in P. The local store σ is a list of pairs of the form $\{x,\widetilde{v}\}$ (see Def. 3). A monitor $s\lfloor S\cdot\widetilde{x}\cdot\widetilde{u}\rfloor$ is identified by the session name s, contains its session type S (see below), a list $\widetilde{x}$ containing all the variables used by the process, and a list of names $\widetilde{u}$ that the process has used in the session. These two lists will be useful to rebuild prefixes. A monitor type T describes the behavior of its associated session. The syntax of types assumes a set of basic *sorts* (bool, int, ...), ranged over U. We also assume $\mathscr{U}$ as the set of all possible values belonging to basic sorts; this way, $\mathscr{V} = \mathscr{N} \cup \mathscr{U}$ is the set of values that processes can exchange. We use v,w (and their decorated versions) to range over $\mathscr{V}$. The type $!U.S$ (resp. $?U.S$) indicates that the owner of the monitor may send (resp. receive) a value of type T and proceed with the behavior prescribed by S.

Types $!U.S$ and $?U.S$ are standard in session types disciplines. A novelty in our work is the (run-time) type $S_1\hat{}S_2$: it indicates that S_1 is the past (already executed) behavior of the associated session, while S_2 represents the present behavior (yet to be executed). That is, the separator $\hat{}$ is used as a cursor in a type; it is inspired by the separator used in [7] to remember the past of sequential CCS processes. These session types with *present and past* occur only at run-time; the intent is that each time that the process performs a forward computation the cursor will be moved forward by one action; it will be moved backwards by one action as result of a reversible action.

The syntax of processes follows standard lines: we consider the idle process $\mathbf{0}$, prefixes for session establishment (noted $u(x:S).P$ and $\overline{u}(x:S).Q$, where S is a session type), and prefixes for intra-session communication (noted $k(x).P$ and $k\langle v\rangle.P$). We write $\mathscr{P}$ and $\mathscr{M}$ to indicate the set of processes and configurations, resp. We call *agent* an element of the set $\mathscr{A} = \mathscr{M} \cup \mathscr{P}$. We let P,Q (and their decorated versions) to range over $\mathscr{P}$; also, we use L,M,N to range over $\mathscr{M}$ and A,B,C to range over $\mathscr{A}$.

Before formally presenting the operational semantics, we give some intuitions on the information

$$(\text{E.ParC})\ A \parallel B \equiv B \parallel A \qquad (\text{E.ParA})\ A \parallel (B \parallel C) \equiv (A \parallel B) \parallel C \qquad (\text{E.NilM})\ A \parallel \mathbf{0} \equiv A$$

$$(\text{E.NewN})\ \nu n.\mathbf{0} \equiv \mathbf{0} \qquad (\text{E.NewC})\ \nu n.\nu m.A \equiv \nu m.\nu n.A \qquad (\text{E.NewP})\ (\nu n.A) \parallel B \equiv \nu n.(A \parallel B)$$

$$(\text{E.}\alpha)\ A =_\alpha B \implies A \equiv B$$

Figure 2: Structural congruence

carried by monitors, in particular the variable lists. Consider the following configuration, with $s \in \delta$:

$$\langle P, \sigma \rangle_\delta \parallel s\lfloor S.?U\hat{}T \cdot \widetilde{x}, x \cdot \widetilde{u}, k \rfloor$$

By inspecting the type before the cursor $\hat{}$, we know the last action of the process, before becoming P, was an input; also, by the additional information in the variable and name lists, we know that this input action was of the form $k(x)$. That is, the shape of the process right before the input action was $k(x).P$.

Operational Semantics. The operational semantics of our reversible calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over configurations $\rightarrow \subset \mathscr{M} \times \mathscr{M}$, and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations $\equiv \subset \mathscr{P}^2 \cup \mathscr{A}^2$.

Definition 1 (Contexts)**.** *Configuration contexts*, also called evaluation contexts, are configurations with one hole "$\cdot$" defined by the following grammar: $\mathbb{E} ::= \cdot \mid (M \parallel \mathbb{E}) \mid \nu n.\mathbb{E}$. General contexts $\mathbb{C}$ are processes or configurations with one hole $\cdot$", and are obtained from processes or configurations by replacing one occurrence of $\mathbf{0}$ (either as process or as configuration) with $\cdot$.

A congruence on processes and configurations is an equivalence relation $\mathscr{R}$ that is closed under general contexts: $P \mathscr{R} Q \implies \mathbb{C}[P] \mathscr{R} \mathbb{C}[Q]$ and $M \mathscr{R} N \implies \mathbb{C}[M] \mathscr{R} \mathbb{C}[N]$. The relation $\equiv$ is defined as the smallest congruence, on processes and configurations, that satisfies rules in Figure 2. In defining the rules we adopt Barendregt's Variable Convention: If terms $t_1, \ldots, t_n$ occur in a certain context (e.g. definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones. This is why in Rule (E.NewP) there is no check on free names.

A binary relation $\mathscr{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$(\text{Ctx})\ \frac{M \mathscr{R} N}{\mathbb{E}[M] \mathscr{R} \mathbb{E}[N]} \qquad (\text{Eqv})\ \frac{M \equiv M' \qquad M' \mathscr{R} N' \qquad N' \equiv N}{M \mathscr{R} N}$$

The reduction relation $\rightarrow$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the backward reduction relation $\rightsquigarrow$: $\rightarrow = \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are the smallest evaluation-closed relations satisfying the rules in Figure 3. Before commenting them we need some definitions in place:

Definition 2 (Dual type)**.** The dual type of a type S, indicated as $\overline{S}$, is inductively defined as follows:

$$\overline{!U.S} = ?U.\overline{S} \qquad \overline{?U.S} = !U.\overline{S} \qquad \overline{\mathtt{end}} = \mathtt{end}$$

Sometimes we will write $\mathsf{dual}(S_1, S_2)$ to indicate $S_1 = \overline{S_2}$.

Remark 1 (Store and Explicit Substitution)**.** One of the main challenges in defining reversible semantics for processes is how to treat substitutions, since in general a substitution is not a bijective function. There are at least two possibilities. First, one may create a copy of a process before applying a substitution and

(OPEN)
$$\frac{\mathsf{dual}(S,T) \qquad \overline{s} \notin \delta \qquad s \notin \delta' \qquad \sigma_1(u) = \sigma_2(u')}{\begin{array}{c}\langle \overline{u}(x:S).P, \sigma_1 \rangle_\delta \parallel \langle u'(y:T).Q, \sigma_2 \rangle_{\delta'} \quad \twoheadrightarrow \\ (\nu s,\overline{s}).\left(\langle P, \sigma_1[x \mapsto \overline{s}] \rangle_{\delta,\overline{s}} \parallel \overline{s}\lfloor \hat{}S \cdot x \cdot \overline{u} \rfloor \parallel \langle Q, \sigma_2[y \mapsto s] \rangle_{\delta',s} \parallel s\lfloor \hat{}T \cdot y \cdot u' \rfloor\right)\end{array}}$$

(OPEN*)
$$\frac{\mathsf{dual}(S,T) \qquad \sigma_1(u) = \sigma_2(u')}{\begin{array}{c}(\nu s,\overline{s}).\left(\langle P, \sigma_1 \rangle_{\delta,\overline{s}} \parallel \overline{s}\lfloor \hat{}S \cdot x \cdot \overline{u} \rfloor \parallel \langle Q, \sigma_2 \rangle_{\delta',s} \parallel s\lfloor \hat{}T \cdot y \cdot u' \rfloor\right) \quad \rightsquigarrow \\ \langle \overline{u}(x:S).P, \sigma_1 \setminus x \rangle_\delta \parallel \langle u'(y:T).Q, \sigma_2 \setminus y \rangle_{\delta'}\end{array}}$$

(COM)
$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(\overline{k}') = \overline{s} \qquad \overline{s} \in \delta' \qquad \sigma_2(y) = v}{\begin{array}{l}\langle k(x).P, \sigma_1 \rangle_\delta \parallel \langle \overline{k}'\langle y \rangle.Q, \sigma_2 \rangle_{\delta'} \parallel s\lfloor T.\hat{}?U.S_1 \cdot \widetilde{x} \cdot \widetilde{n} \rfloor \parallel \overline{s}\lfloor T'.\hat{}!U.S_2 \cdot \widetilde{y} \cdot \widetilde{m} \rfloor \quad \twoheadrightarrow \\ \langle P, \sigma_1[x \mapsto v] \rangle_\delta \parallel \langle Q, \sigma_2 \rangle_{\delta'} \parallel s\lfloor T.?U.\hat{}S_1 \cdot \widetilde{x},x \cdot \widetilde{n},k \rfloor \parallel \overline{s}\lfloor T'.!U.\hat{}S_2 \cdot \widetilde{y},y \cdot \widetilde{m},k' \rfloor\end{array}}$$

(COM*)
$$\frac{\sigma_1(k) = s \qquad s \in \delta \qquad \sigma_2(\overline{k}') = \overline{s} \qquad \overline{s} \in \delta'}{\begin{array}{l}\langle P, \sigma_1 \rangle_\delta \parallel \langle Q, \sigma_2 \rangle_{\delta'} \parallel s\lfloor T.?U.\hat{}S_1 \cdot \widetilde{x},x \cdot \widetilde{n},k \rfloor \parallel \overline{s}\lfloor T'.!U.\hat{}S_2 \cdot \widetilde{y},y \cdot \widetilde{m},k' \rfloor \quad \rightsquigarrow \\ \langle k(x).P, \sigma_1 \setminus x \rangle_\delta \parallel \langle \overline{k}\langle y \rangle.Q, \sigma_2 \rangle_{\delta'} \parallel s\lfloor T.\hat{}?U.S_1 \cdot \widetilde{x} \cdot \widetilde{n} \rfloor \parallel \overline{s}\lfloor T'.\hat{}!U.S_2 \cdot \widetilde{y} \cdot \widetilde{m} \rfloor\end{array}}$$

Figure 3: Operational Semantics: Forward ($\twoheadrightarrow$) and Backward ($\rightsquigarrow$) Reduction Semantics.

then replace the process with its copy when reverting the substitution; this is the strategy developed in [23]. Second, one may use a store and a mechanism with explicit substitution, following [24, 10]. The first technique creates a memory each time a value is substituted; here we implement the second technique, in which one just has to remember the pair variable/value for each substitution.

We now formally define the store σ and the operations on it.

Definition 3. A local store σ is a mapping from variables to an ordered list of values. Given a store σ, a variable x, and a value v, we define the *update* $\sigma[x \mapsto v]$ and *reverse update* $\sigma \setminus x$ as follows:

$$\sigma[x \mapsto v] = \begin{cases} \sigma \cup \{x,v\} & \text{if } x \notin dom(\sigma) \\ \sigma_1 \cup \{x, \widetilde{v} \cdot v\} & \text{if } \sigma = \sigma_1 \cup \{x, \widetilde{v}\} \end{cases} \qquad \sigma \setminus x = \begin{cases} \sigma_1 & \text{if } \sigma = \sigma_1 \cup \{x,v\} \\ \sigma_1 \cup \{x, \widetilde{v}\} & \text{if } \sigma = \sigma_1 \cup \{x, \widetilde{v} \cdot v\} \end{cases}$$

The *evaluation* of name n under a local store σ, written $\sigma(n)$, is the value v if $\{n,v\} \in \sigma$ or $\{n, \widetilde{v} \cdot v\} \in \sigma$; otherwise, it is n itself. Let us stress the fact that our store maintains a correspondence between variables and *lists* of values in order to enable reversibility. The list represents at a given time the assignment history of the variable to which it corresponds, with the actual value of the variable being at the top of the list. If $\sigma(n) = n$ then n is not a variable.

The rules of the operational semantics are in Fig. 3. We briefly comment on them:

- Rule OPEN is the forward rule for session establishment. It creates two fresh, dual endpoints and their associated monitors. Each monitor stores a session type; each store records the name of the fresh session endpoints and of the channel. The monitor records the name on which the session has started and the variable used by the process to refer to the endpoint. Establishing a new session requires type duality and that the two process refer to the same name (cf. condition $\sigma_1(u) = \sigma_2(u')$).

- Rule OPEN* is the exact opposite of Rule OPEN. In order to revert a session creation, the rule checks that session types are at their initial position (e.g., $\hat{}S$ and $\hat{}T$). Moreover, the variable and name lists should contain just one element each. The effect of the rule is to collect the two endpoint names and to eliminate the two associated monitors, while restoring the prefixes in the processes.
- Rule COM describes intra-session communication. Two running processes can communicate if they refer to the same session. The sent value v is obtained by evaluating the sent value y under the sender store σ_2. The result of a communication is that the store of the receiver is updated with a new value, bound to the read variable: $\sigma_1[x \mapsto v]$. Also, both monitor types are moved one step forward, and both read and sent variables are put on the top of the list in their respective monitors; the same occurs for the session names. This way we keep information about the prefixes (i.e., $k(x)$ and $\overline{k'}\langle y\rangle$).
- Rule COM* undoes a communication: the sent value (along with the variable used to read it) is eliminated from the store of the receiving process. The variable list of the monitor keeps information on which variable to unbound: indeed, it is the variable at the top of this list the one that has to be eliminated, as we want to revert the input of its associated value. Moreover, information contained in the name list of the monitors is used to recover output and input prefixes. Notice that the information about the kind of prefix to be built again (input or output) is given by the type of the monitor. A further consequence of undoing a communication is that the session types are moved one step backward.

Our process framework satisfies the so-called *loop lemma*, a property that gives us a basic guarantee of the consistency between forward and backward reductions. We require the following definition:

Definition 4 (Initial and Reachable Configurations)**.** A configuration is *initial* if there are no monitors and all the running processes have an empty store and are identified by $\emptyset$. A configuration is *reachable* if it can be derived using $\rightarrow$ from an initial configuration.

An easy induction on the structure of terms provides us with a kind of normal form for configurations:

Lemma 1 (Normal Form)**.** *For any configuration M we have that:*

$$M \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\langle P_i,\sigma_i\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{y}_j\cdot\widetilde{u}_j\rfloor\right)$$

Notice that, by convention, we assume that $\prod_{i\in I}A_i = \mathbf{0}$ if $I=\emptyset$. Then we have:

Lemma 2 (Loop Lemma)**.** *For any reachable configuration M,N, we have $M \twoheadrightarrow N \iff N \rightsquigarrow M$.*

Proof (Sketch). By induction on the derivation of $M \twoheadrightarrow N$ for the if direction, and on the derivation of $N \rightsquigarrow M$ for the converse. We will just show the forward case when the applied rule is OPEN; the other cases are similar. By Lemma 1 we have that:

$$M \equiv \nu\widetilde{a}.\left(\prod_{i\in I}\langle P_i,\sigma_i\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{y}_j\cdot\widetilde{u}_j\rfloor\right)$$

and since rule OPEN is applied, then there exist two indexes $w,z\in I$ such that $P_w = \overline{u}(x:S).Q_w$ and $P_z = u'(y:T).Q_z$ with $\mathsf{dual}(S,T)$, $\overline{s}\notin\delta_w$, $s\notin\delta_z$ and $\sigma_1(u)=\sigma_2(u')$. We have then:

$$M \twoheadrightarrow \nu\widetilde{a},s,\overline{s}.\big(\prod_{i\in I'}\langle P_i,\sigma_i\rangle_{\delta_i} \parallel \prod_{j\in J} s_j\lfloor H_j\cdot\widetilde{y}_j\cdot\widetilde{u}_j\rfloor \parallel \langle P_w,\sigma_w[x\mapsto\overline{s}]\rangle_{\delta_w,\overline{s}} \parallel \langle P_z,\sigma_z[y\mapsto s]\rangle_{\delta_z,s} \parallel s_w\lfloor\hat{}T\cdot x\cdot\overline{u}\rfloor \parallel s_z\lfloor\hat{}S\cdot y\cdot u\rfloor\big)$$

with $I' = I\setminus\{w,z\}$. It is easy to see that by applying OPEN* we get back to M, as desired. □

Another property that a reversible calculus should enjoy is the so-called *square lemma*, which may be informally described as follows. Assume a configuration from which two reductions are possible: if these reductions are *concurrent* then the order in which the two reductions are applied does not matter, and the same configuration is reached. This lemma therefore relies on the definition of concurrent transitions. In our setting, thanks to the information carried by the monitors and to linearity of sessions, we may decree that two reductions are concurrent if they operate on different sessions/channels (service names). One may instrument the reduction semantics $\rightarrow$ with a label λ containing the endpoints used by the rule and the service name (if any): $M \xrightarrow{\lambda} N$. Then, reductions $M \xrightarrow{\lambda_1} N$ and $M \xrightarrow{\lambda_2} N$ are concurrent if $\lambda_1 \cap \lambda_2 = \emptyset$.

Using the square lemma one may then show that the reversible semantics is *causally consistent*, i.e., that an action can be reverted only after all the actions causally depending on it have already been reverted. We leave for future work establishing these further results for our framework.

3 Concluding Remarks and Future Work

We have proposed a fresh approach to reversible semantics for session processes: it builds upon a style of process semantics in which monitors (which include session types) enable process reductions. Even if this style of process semantics is not new—it was introduced in [19] and later used in [17, 21, 13, 9]—to our knowledge this is the first time that this formulation is used to support reversibility.

We rely on monitors which contain types that describe past and future structured interactions; these types offer a natural form of memories for supporting forward and backward semantics. We motivated our approach by introducing a simple process framework with session establishment and communication; extensions with other usual session constructs (labeled choice and recursion) are straightforward. To highlight the simplicity of our approach, we have considered binary session types. We believe that our approach scales to account for multiparty structured communications; in such a setting, monitors would be generated after multiparty session establishment, and would be equipped with local projections of global types, as in [2, 8]. A multiparty, asynchronous semantics may need to consider forms of *coordinated* reversibility among different partners; we plan to address these challenges in future work.

Most models of reversible processes (cf. [11]) do not consider (behavioral) types, and so their reversible semantics must account for arbitrary forms of concurrent behavior. In reversing the untyped π-calculus, substitutions and scope extrusion are known to be challenging issues [10]. Reversing session processes is a seemingly simpler problem, as behavior is disciplined by types: once a session is established, concurrency interactions proceed in a deterministic, confluent manner. Also, in session π-calculi scope extrusion is limited. To our knowledge, the work [25] is the first to address reversibility for a synchronous π-calculus with binary session types. A key difference between our work and [25] is the role that session types play in the reversible semantics. We have used session types to define forward and backward semantics for session processes; in contrast, the reversible semantics in [25] establishes key results for reversibility (e.g., the square lemma and causal consistency) using an untyped reduction semantics. Hence, although the reversible session π-calculus in [25] is shown to be typable using standard binary session types, the influence of types on the reversible semantics of [25] is indirect at best.

As further topics for future work, we plan to establish the precise savings involved from moving from (i) an untyped reversible semantics to (ii) a monitored reversible semantics with types, as proposed here. We plan to compare the (untyped) reversible higher-order processes in [23] and the core higher-order session calculus in [20], which may precisely encode the first-order session π-calculus. Another direction concerns *controlled reversibility* [22]. Some recent approaches have proposed types with controlled roll-back and explicit checkpoints among parties [1]. We believe that by adding explicit rollback into

(run-time) type information, we could achieve intuitive mechanisms for controlled reversibility.

Acknowledgments. We are grateful to Dimitris Kouzapas for useful exchanges. We would also like to thank the anonymous reviewers for their suggestions, which were helpful to improve the presentation. Pérez is also affiliated to the NOVA Laboratory for Computer Science and Informatics (NOVA LINCS - PEst/UID/CEC/04516/2013), Universidade Nova de Lisboa, Portugal.

References

[1] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese & Ugo de'Liguoro (2016): *Retractable Contracts*. In Simon Gay & Jade Alglave, editors: *PLACES 2015, Electronic Proceedings in Theoretical Computer Science* 203, Open Publishing Association, pp. 61–72, doi:10.4204/EPTCS.203.5.

[2] Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda & Nobuko Yoshida (2013): *Monitoring Networks through Multiparty Session Types*. In: *Proc. of FMOODS/FORTE 2013, Lecture Notes in Computer Science* 7892, Springer, pp. 50–65, doi:10.1007/978-3-642-38592-6_5.

[3] Luís Caires, Carla Ferreira & Hugo Torres Vieira (2009): *A Process Calculus Analysis of Compensations*. In: *TGC 2008, LNCS* 5474, Springer, pp. 87–103, doi:10.1007/978-3-642-00945-7_6.

[4] Luís Caires & Hugo Torres Vieira (2010): *Conversation types*. *Theor. Comput. Sci.* 411(51-52), pp. 4399–4440, doi:10.1016/j.tcs.2010.09.010.

[5] Sara Capecchi, Ilaria Castellani & Mariangiola Dezani-Ciancaglini (2011): *Information Flow Safety in Multiparty Sessions*. In: *Proc. of EXPRESS 2011, EPTCS* 64, pp. 16–30, doi:10.4204/EPTCS.64.2.

[6] Sara Capecchi, Elena Giachino & Nobuko Yoshida (2016): *Global escape in multiparty sessions*. *Mathematical Structures in Computer Science* 26(2), pp. 156–205, doi:10.1017/S0960129514000164.

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

[8] Ilaria Castellani, Mariangiola Dezani-Ciancaglini & Jorge A. Pérez (2014): *Self-Adaptation and Secure Information Flow in Multiparty Structured Communications: A Unified Perspective*. In: *BEAT 2014, EPTCS* 162, pp. 9–18, doi:10.4204/EPTCS.162.2.

[9] Mario Coppo, Mariangiola Dezani-Ciancaglini & Betti Venneri (2015): *Self-adaptive multiparty sessions*. *Service Oriented Computing and Applications* 9(3-4), pp. 249–268, doi:10.1007/s11761-014-0171-9.

[10] Ioana Cristescu, Jean Krivine & Daniele Varacca (2013): *A Compositional Semantics for the Reversible p-Calculus*. In: *Proc. of LICS2013*, IEEE Computer Society, pp. 388–397, doi:10.1109/LICS.2013.45.

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

[12] Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova & Nobuko Yoshida (2015): *Practical interruptible conversations: distributed dynamic verification with multiparty session types and Python*. *Formal Methods in System Design* 46(3), pp. 197–225, doi:10.1007/s10703-014-0218-8.

[13] Cinzia Di Giusto & Jorge A. Pérez (2014): *An Event-Based Approach to Runtime Adaptation in Communication-Centric Systems*. Research Report, Universite de Nice Sophia-Antipolis (UNS) ; University of Groningen. Available at https://hal.archives-ouvertes.fr/hal-01093090. To appear in Post-proc. of WS-FM 2014 (Springer LNCS).

[14] Cinzia Di Giusto & Jorge A. Pérez (2015): *Disciplined structured communications with disciplined runtime adaptation*. *Sci. Comput. Program.* 97, pp. 235–265, doi:10.1016/j.scico.2014.04.017.

[15] Kohei Honda, Vasco Thudichum Vasconcelos & Makoto Kubo (1998): *Language Primitives and Type Discipline for Structured Communication-Based Programming*. In: *ESOP'98, LNCS* 1381, Springer, pp. 122–138, doi:10.1007/BFb0053567.

[16] Kohei Honda, Nobuko Yoshida & Marco Carbone (2008): *Multiparty asynchronous session types*. In: *POPL 2008*, ACM, pp. 273–284, doi:10.1145/1328438.1328472.

[17] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida & Kohei Honda (2010): *Type-Safe Eventful Sessions in Java*. In: *Proc. of ECOOP 2010, LNCS* 6183, Springer, pp. 329–353, doi:10.1007/978-3-642-14107-2_16.

[18] Limin Jia, Hannah Gommerstadt & Frank Pfenning (2016): *Monitors and blame assignment for higher-order session types*. In: *POPL 2016*, ACM, pp. 582–594, doi:10.1145/2837614.2837662.

[19] Dimitrios Kouzapas (2009): *A Session Type Discipline for Event Driven Programming Models*. Master's thesis, Imperial College London. Available at http://www.doc.ic.ac.uk/teaching/distinguished-projects/2009/d.kouzapas.pdf.

[20] Dimitrios Kouzapas, Jorge A. Pérez & Nobuko Yoshida (2016): *On the Relative Expressiveness of Higher-Order Session Processes*. In: *ESOP 2016*, LNCS, Springer. To appear.

[21] Dimitrios Kouzapas, Nobuko Yoshida & Kohei Honda (2011): *On Asynchronous Session Semantics*. In: *Proc. of FMOODS 2011 and FORTE 2011, LNCS* 6722, Springer, pp. 228–243, doi:10.1007/978-3-642-21461-5_15.

[22] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2011): *Controlling Reversibility in Higher-Order Pi*. In: *Proc. of CONCUR 2011*, LNCS, Springer, pp. 297–311, doi:10.1007/978-3-642-23217-6_20.

[23] Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2010): *Reversing Higher-Order Pi*. In: *Proc. of CONCUR 2010*, LNCS, Springer, pp. 478–493, doi:10.1007/978-3-642-15375-4_33.

[24] Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina & Jean-Bernard Stefani (2012): *A Reversible Abstract Machine and Its Space Overhead*. In: *Proc. of FMOODS/FORTE 2012*, LNCS, Springer, pp. 1–17, doi:10.1007/978-3-642-30793-5_1.

[25] Francesco Tiezzi & Nobuko Yoshida (2015): *Reversible session-based pi-calculus*. *J. Log. Algebr. Meth. Program.* 84(5), pp. 684–707, doi:10.1016/j.jlamp.2015.03.004.