Encoding Choice and Replication in roll-π

Adam D. Barwell[1][0000−0003−1236−7160], Ping Hou[2][0000−0001−6899−9971], Martin Vassor[3][0000−0002−2057−0495], and Nobuko Yoshida[2][0000−0002−3925−8557]

[1] School of Computer Science, University of St Andrews, Scotland, UK
[2] Department of Computer Science, University of Oxford, England, UK
[3] Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract. Reversible process calculi, such as **roll**-π, can accommodate fault-tolerant communication protocols. However, untyped process calculi cannot guarantee desirable behavioural properties such as deadlock-freedom. Concomitantly, Multiparty Session Types (MPST) can provide such guarantees by construction, but often do not consider failures. This suggests that, by leveraging both MPST and **roll**-π, we can facilitate the representation of failures without requiring a significant extension to the MPST theory. However, **roll**-π lacks choice and replication primitives, which are key features for an MPST-based type system. Nonetheless, its expressiveness allows these constructs to be encoded directly. In this paper, we introduce **roll**-$\pi!\oplus$, a variant of **roll**-π that incorporates choice and replication, and outline its encoding in **roll**-π. This extension lays the foundation for integrating **roll**-π with MPST.

Keywords: Reversible Process Calculus · Causally Consistent Rollback · Multiparty Session Types · Fault-Tolerance

1 Introduction

Motivation. Multiparty session types (MPST) [3] are a family of type systems for message-passing concurrent programs, guaranteeing that well-typed programs are free from errors such as deadlocks and protocol violations. Generally, MPST consist of two constructs: *communication* and *recursion*. During communication, the sender transmits one label from a set of allowed labels to the receiver. The selected label determines how the protocol proceeds, providing a form of *choice*.

Consider the following representation of Sangiorgi's well-known *vending machine* example [12] in the first-order π-calculus, illustrating these key concepts.

$$!coin(x).(request_tea(y).\overline{tea}\langle z\rangle \;+\; request_coffee(y).\overline{coffee}\langle z\rangle) \tag{1}$$

Following insertion of a coin, the buyer chooses either *tea* or *coffee*, which is then delivered. This process can be extended to repeat after a successful vend.

MPST usually assume a reliable communication model, where processes and communication channels are fault-free. In such a model, message exchanges, choices, and recursion are mapped to the (forward only, first-order) π-calculus,

using its primitives for messages, choices, and replication. Some MPST systems relax these reliability assumptions to accommodate real-world settings, leading to more complex frameworks, such as Affine MPST [4], MPST with explicit checkpoints [2], and MPST with explicit (un)reliable communications [11].

Reversible high-order calculi, such as the **roll**-π calculus [5,6], are capable of modeling communication protocols that tolerate faults [7, 10, 14, 15]. In **roll**-π, communication is represented by primitives including $c\langle P\rangle$ (sending a *pending message*) and $c(X) \triangleright_\gamma (Q)$ (a *trigger* to receive a message, where X and γ are bound in Q). Processes are annotated with *keys* (e.g. k), which serve as rollback points and track causal dependencies. Unlike first-order π-calculi, messages in **roll**-π consist of processes rather than names. Upon reception, free occurrences of X are substituted with P in Q, and a fresh key is generated to replace free occurrences of γ. Additionally, a memory is created, storing the trigger and pending message before substitution, to enable recovery during rollback. Rollbacks are triggered by the primitive $\texttt{roll}\ \gamma$. This behaviour can be illustrated as follows:

$$k_1 :: \mathit{coin}\langle 0\rangle \mid\mid k_2 :: \mathit{coin}(X) \triangleright_\gamma (\texttt{roll}\ \gamma)$$

$$\twoheadrightarrow \nu k.(\texttt{roll}\ k \mid\mid [k_1 :: \mathit{coin}\langle 0\rangle \mid\mid k_2 :: \mathit{coin}(X) \triangleright_\gamma (\texttt{roll}\ \gamma); k])$$

$$\overset{2}{\rightsquigarrow} k_1 :: \mathit{coin}\langle 0\rangle \mid\mid k_2 :: \mathit{coin}(X) \triangleright_\gamma (\texttt{roll}\ \gamma)$$

A consequence of these features is that **roll**-π can naturally model communication protocols with faults: its reversible structure explicitly tracks the causal history of communications, allowing processes to roll back to earlier states when errors or unexpected behaviours occur.

Objective. We aim to leverage **roll**-π to enable transparent fault recovery in MPST. To that end, we propose the following steps: *1)* use **roll**-π as the framework for a multiparty session calculus, **st-roll**-π; *2)* develop a type system for **st-roll**-π and investigate whether typed **roll**-π processes ensure desirable properties while accounting for reversibility; and *3)* integrate MPST with failure models, e.g. crash-recovery, to enable **st-roll**-π-based MPST to support various fault recovery mechanisms.

This paper takes a preliminary step towards our objective by introducing **roll**-$\pi!\oplus$, which extends **roll**-π with choice and replication, and forms a foundation for integrating **roll**-π with MPST for fault recovery. We agree with Mezzina's claim that such constructs can be encoded in **roll**-π [6], and aim to prove this result. While these layers are, in principle, independent of one another, for clarity, we first introduce replication into **roll**-π, followed by the addition of choice to **roll**-π with replication. The **roll**-π grammar, along with the key extensions of **roll**-$\pi!\oplus$, is presented in Fig. 1.

$$
\begin{array}{lll}
\mathcal{P},\mathcal{Q} & ::= 0 \mid X \mid \nu c.\,(\mathcal{P}) \mid \mathcal{P} \parallel \mathcal{Q} & \textbf{roll-}\pi \textit{ processes} \\
& \quad \mid c\langle \mathcal{P} \rangle \mid c(X) \triangleright_\gamma (\mathcal{P}) \mid \texttt{roll}\ \gamma \mid \texttt{roll}\ k & \\
& \quad \mid !^r \mathcal{P} & \textit{Replication} \\
& \quad \mid \mathcal{P} \oplus_\gamma \mathcal{Q} & \textit{Choice} \\
\mathcal{M},\mathcal{N} & ::= 0 \mid \nu u.\,(\mathcal{M}) \mid \mathcal{M} \parallel \mathcal{N} \mid k : P & \textbf{roll-}\pi \textit{ configurations} \\
& \quad \mid [\mu; k] \mid [\mu; k]^\bullet & \\
\kappa & ::= k \mid \langle h, \tilde{h} \rangle \cdot k & \textit{Simple and Complex Tags} \\
\mu & ::= \kappa_1 : c\langle \mathcal{P} \rangle \parallel \kappa_2 : a(X) \triangleright_\gamma (\mathcal{Q}) & \textit{Communication memory} \\
& \quad \mid \kappa : !^r \mathcal{P} & \textit{Replication memory} \\
& \quad \mid \kappa : \mathcal{P} \oplus_\gamma \mathcal{Q} & \textit{Choice memory}
\end{array}
$$

Fig. 1. roll-π and its extensions, with replication and choice additions in blue and red, respectively ($c \in \mathbb{N}$: channel names; $k \in \mathbb{K}$: keys; $X \in \mathbb{V}_p$: process variables; $\gamma \in \mathbb{V}_k$: key variables; $u \in \mathbb{N} \cup \mathbb{K}$: identifiers).

2 roll-π with Replication: roll-π!

In this section, we introduce **roll-π!**, a variant of **roll-π** with a replication operator, and illustrate its encoding into **roll-π**. In the (forward-only) **HOπ** calculus, replicating a process P is straightforward:

$$!P = r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \parallel r\langle r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \rangle$$

$$\rightarrow r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \parallel r\langle r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \rangle \parallel P = !P \parallel P$$

We denote $r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \parallel r\langle r(X) \triangleright (X \parallel r\langle X \rangle \parallel P) \rangle$ as $\mathbb{R}(P, r)$.

Remark 1. When replication is a primitive, it is generally understood as a structural congruence [8]: $!P \equiv !P \parallel P$. Conversely, we implement replication through communication, requiring a computation step (i.e. a reduction) to be performed. This may impact, e.g. *termination*. For simplicity, we disregard this difference by distinguishing between replication and non-replication reductions and considering termination up to replication reductions.

To implement replication, we incorporate the **HOπ** term into **roll-π**, where reduction generates additional memories. Thus, the replication operator must be defined whilst accounting for these memories and ensuring the isolation of the auxiliary *administrative* channel (e.g. r). We address this in two steps: *1)* we extend **roll-π** with *explicit-channel replication*, where the replication operator specifies the administrative channel ($!^r P$), and show its encodability in **roll-π**; and *2)* we define a variant without explicit channel annotations and demonstrate its encodability into $!^r P$ and, by transitivity, into **roll-π**.

2.1 roll-π!

Syntax. We extend the syntax of **roll-π** with the two new rules (in blue in Fig. 1), introducing $!^r P$ in both processes and memories; where r is not free in P.

Semantics. Structural congruence remains unchanged; specifically, $!P \not\equiv !P \parallel P$. We instead introduce (Repl.E) to replicate processes.

$$(\text{Repl.E})\ \frac{\mu = \kappa : {!^r P}}{\kappa : {!^r P} \twoheadrightarrow \nu k.\,(k : P \parallel {!^r P} \parallel [\mu; k])}$$

Example 1. With replication, we can easily encode a restricted version of (1) where no beverage choice is available (see Ex. 2 in § 3 for the full version).

$$\begin{array}{l} \mathit{machine} : {!\mathit{coin}(X) \triangleright (\mathit{request}(X) \triangleright (\mathit{beverage}\langle 0\rangle))} \\ \parallel \mathit{customer} : \mathit{coin}\langle 0\rangle \parallel \mathit{request}\langle 0\rangle \parallel \mathit{beverage}(X) \triangleright (0) \end{array}$$

$$\rightarrow\ \nu \ldots. \begin{pmatrix} \langle m_1, \tilde{m}\rangle \cdot m : \mathit{coin}(X) \triangleright (\mathit{request}(X) \triangleright (\mathit{beverage}\langle 0\rangle)) \\ \parallel \langle m_2, \tilde{m}\rangle \cdot m : {!\mathit{coin}(X) \triangleright (\mathit{request}(X) \triangleright (\mathit{beverage}\langle 0\rangle))} \\ \parallel \mathit{customer} : \mathit{coin}\langle 0\rangle \parallel \mathit{request}\langle 0\rangle \parallel \mathit{beverage}(X) \triangleright (0) \\ \parallel \ldots \end{pmatrix}$$

$$\rightarrow^{\star}\ \nu \ldots. \begin{pmatrix} \langle m_2, \tilde{m}\rangle \cdot m : {!\mathit{coin}(X) \triangleright (\mathit{request}(X) \triangleright (\mathit{beverage}\langle 0\rangle))} \\ \parallel \ldots \end{pmatrix}$$

Replication is compatible with rollbacks: providing an adequate `roll` (not shown here), the process could reduce backward to its initial state:

$$\rightsquigarrow^{\star}\ \begin{array}{l} \mathit{machine} : {!\mathit{coin}(X) \triangleright (\mathit{request}(X) \triangleright (\mathit{beverage}\langle 0\rangle))} \\ \parallel \mathit{customer} : \mathit{coin}\langle 0\rangle \parallel \mathit{request}\langle 0\rangle \parallel \mathit{beverage}(X) \triangleright (0) \end{array}$$

2.2 Encoding roll-π! in roll-π.

We restrict our encoding (denoted $[\![\cdot]\!]_{!^c}$) to terms where ${!^r P}$ appears only in restricted form (i.e. as $\nu r.\,({!^r P})$) and r does not occur free in the nested process P. These constraints are easily enforced in practice. Given these conditions, consistent **roll**-π! configurations are easily encoded into **roll**-π: $[\![\cdot]\!]_{!^c}$ recursively encodes occurrences of ${!^r P}$ as $\mathbb{R}([\![Q]\!]_{!^c}, r)$, including in memories.

Implicit channels. In § 2.1, replication is explicitly annotated with the channel used in encoding. However, these names do not play an active role in the language. Therefore, we define a variant of **roll**-π! where channel annotations are erased and show that it can be encoded in **roll**-π!.

Results. We first illustrate a strong barbed bisimulation between **roll**-π and **roll**-π with explicit-channel replication. Given the relation $\mathcal{R} = \{\langle M, [\![M]\!]_{!^c}\rangle\}$, which associates each configuration with its encoding, we demonstrate that $\equiv \mathcal{R} \equiv$ is a strong barbed bisimulation. Next, we relate terms of **roll**-π with replication, both with and without explicit channels (via an erasure function), showing that this relation also forms a strong barbed bisimulation. This step is straightforward, as the replication channel annotation in $\mathbb{R}(P, r)$ is used only for encoding and carries no semantics implications. Combining these bisimulations, we conclude that basic **roll**-π and **roll**-π! are strongly barbed bisimilar.

3 roll-π! with Choice: roll-π!⊕

In this section, we present **roll-π!⊕**, which extends **roll-π!** from § 2 with a choice operator. A process can perform a choice between two processes, P and Q, by guarding them on a single channel and sending one message to release only one of the processes. This leads to the naïve implementation of choice:

$$P \oplus_\gamma Q = \nu c.\,(c\langle 0\rangle \parallel c(X) \triangleright_\gamma (P) \parallel c(X) \triangleright_\gamma (Q))$$

$$\twoheadrightarrow \nu c, k.\,\left(P\{^k/_\gamma\} \parallel c(X) \triangleright_\gamma (Q) \parallel [\mu; k]\right)$$

$$\text{or} \quad \twoheadrightarrow \nu c, k.\,\left(c(X) \triangleright_\gamma (P) \parallel Q\{^k/_\gamma\} \parallel [\mu; k]\right)$$

where c and X are not free in P or Q. As with message exchanges, the choice binding γ can be reverted at the user's discretion. We note that the leftover $c(X) \triangleright (P)$ (resp. $c(X) \triangleright (Q)$) is blocked, which can complicate correctness proofs. Additionally, this form of choice is *internal*: the selection is not constrained by the environment, although reversibility allows for a rollback if an unsuitable branch is taken.

Suppose P (resp. Q) can receive on channel p (resp. q). We implement an *external* choice between P and Q as:

$$(P \parallel q(X) \triangleright (\texttt{roll}\ \gamma)) \oplus_\gamma (Q \parallel p(X) \triangleright (\texttt{roll}\ \gamma))$$

Should the environment select the Q branch, by sending on q, when the choice selects the other branch, then the trigger on q would initiate a rollback of the choice, allowing for the correct branch to be selected. Therefore, we focus only on internal choice, ignoring these additional $\texttt{roll}\ \gamma$.

In contrast to the naïve implementation, our approach includes three communications on c in two stages. The first stage sets a rollback point (γ), and the second performs the choice: one of P or Q is sent and discarded (avoiding leftovers), followed by the other process being sent and executed. Note that we include $q(X) \triangleright (\texttt{roll}\ \gamma)$ (resp. $p(X) \triangleright (\texttt{roll}\ \gamma)$) alongside P (resp. Q). This enables external choice: if P is selected but the environment expects a message for Q on q, the message triggers the $\texttt{roll}\ \gamma$, reverting the choice.

Since external choices can be obtained from internal choices via reversibility, we focus only on the internal choice formulation (see (Choice) below). We denote $\nu c.\,(c\langle 0\rangle \parallel c(X) \triangleright_\gamma (c(X) \triangleright (c(X) \triangleright (X)) \parallel c\langle P\rangle \parallel c\langle Q\rangle))$ as $\mathbb{C}(P, Q, \gamma, c)$.

Remark 2. Barbanera et al. [1] studies the relationship between various forms of choices in the context of reversible contracts, particularly focusing on internal and external choices. However, their notion of reversibility differs from ours: when rolling back a choice, the previously selected branch is discarded, resulting in rollback semantics that are not causally consistent. In contrast, our goal is to encode our results within **roll-π**, where maintaining causal consistency is essential.

3.1 roll-π!⊕

Syntax. We extend the syntax of **roll**-π! to accommodate for choices, integrating memory to track the non-deterministic process. Choices are thus incorporated into both processes and memories (highlighted in red in Fig. 1).

Semantics. We extend structural congruence to allow commuting the two sides of a choice and add the following rule to the **roll**-π! semantics in § 2.

$$(\textsc{Choice})\ \frac{\mu = \kappa : \ P \oplus_\gamma Q}{\kappa : \ P \oplus_\gamma Q \twoheadrightarrow \nu k.\left(k : \ P\{^k/_\gamma\ \}\right) \ ||\ [\mu; k]}$$

Example 2. We now fully implement (1). Initially, the machine chooses tea (waiting on channel t), while the customer opts for coffee (sending on channel c). Rollback allows a simple reversal of the choice, selecting coffee instead.

$$\begin{array}{l} \quad \begin{array}{l} \mathit{machine} : \ !\mathit{coin}(X) \triangleright \begin{pmatrix} (c(X) \triangleright (c\langle 0\rangle) \ ||\ t(X) \triangleright (\texttt{roll}\ \gamma)) \\ \oplus_\gamma\, (t(X) \triangleright (t\langle 0\rangle) \ ||\ c(X) \triangleright (\texttt{roll}\ \gamma)) \end{pmatrix} \\ ||\ \mathit{customer} : \ \mathit{coin}\langle 0\rangle \ ||\ c\langle 0\rangle \ ||\ c(X) \triangleright (0) \end{array} \\ \rightarrow^\star \ \nu \ldots. \begin{pmatrix} \langle k_1, \tilde{k}\rangle \cdot k : \ t(X) \triangleright (t\langle 0\rangle) \ ||\ m_4 : \ \texttt{roll}\ k \\ ||\ \langle m_2, \tilde{m}\rangle \cdot m : \ !\mathit{coin}(X) \triangleright (\ldots) \\ ||\ \langle c_3, \tilde{c}\rangle \cdot \mathit{customer} : \ c(X) \triangleright (0) \\ ||\ \left[m_3 : \begin{array}{l} c(X) \triangleright (c\langle 0\rangle) \ ||\ t(X) \triangleright (\texttt{roll}\ \gamma) \\ \oplus_\gamma\, t(X) \triangleright (t\langle 0\rangle) \ ||\ c(X) \triangleright (\texttt{roll}\ \gamma) \end{array}; k\right] \ ||\ \ldots \end{pmatrix} \\ \rightsquigarrow \ \ \nu \ldots. \begin{pmatrix} m_3 : \begin{array}{l} (c(X) \triangleright (c\langle 0\rangle) \ ||\ t(X) \triangleright (\texttt{roll}\ \gamma)) \\ \oplus_\gamma\, (t(X) \triangleright (t\langle 0\rangle) \ ||\ c(X) \triangleright (\texttt{roll}\ \gamma)) \end{array} \\ ||\ \langle m_2, \tilde{m}\rangle \cdot m : \ !\mathit{coin}(X) \triangleright (\ldots) \\ ||\ \langle c_2, \tilde{c}\rangle \cdot \mathit{customer} : \ c\langle 0\rangle \\ ||\ \langle c_3, \tilde{c}\rangle \cdot \mathit{customer} : \ c(X) \triangleright (0) \\ ||\ \ldots \end{pmatrix} \\ \rightarrow^\star \ \nu \ldots. \begin{pmatrix} m_5 : \ 0 \ ||\ \langle k_2, \tilde{k}\rangle \cdot k : \ t(X) \triangleright (\texttt{roll}\ k) \\ ||\ \langle m_2, \tilde{m}\rangle \cdot m : \ !\mathit{coin}(X) \triangleright (\ldots) \\ ||\ \ldots \end{pmatrix} \end{array}$$

3.2 Encoding roll-π!⊕ into roll-π!

To encode **roll**-π!⊕ into **roll**-π!, a naïve approach replaces each occurrence of $P \oplus_\gamma Q$ with $\mathbb{C}(P, Q, \gamma, c)$, for some c not appearing freely in either P or Q. However, this naïve encoding introduces complications with tags. Consider:

$$k : \ P \ ||\ L \oplus_\gamma R \equiv \nu \ldots. \left(\langle h_1, \tilde{h}\rangle \cdot k : \ P \ ||\ \langle h_2, \tilde{h}\rangle \cdot k : \ L \oplus_\gamma R\right) \text{ with } \tilde{h} = \{h_1, h_2\}.$$

Upon reduction, this produces a memory $\left[\langle h_2, \tilde{h}\rangle \cdot k : \ L \oplus_\gamma R; k'\right]$ for a new key k'. Naïvely substituting $L \oplus_\gamma R$ with $\mathbb{C}(P, Q, \gamma, c)$ leads to an incorrect configuration[4], as the memory encoding would be $\left[\langle h_2, \tilde{h}\rangle \cdot k : \ c\langle 0\rangle \ ||\ c(X) \triangleright_\gamma (\ldots); k'\right]$,

[4] Ignoring the restriction $\nu c.\,(\ldots)$, which is easy to address in practice.

which is syntactically invalid. To address this, we must encode $\kappa : L \oplus_\gamma R$ as $\nu c.\,(\kappa_1 : c\langle 0\rangle \mid\mid \kappa_2 : \ldots)$, resulting in a *non*-compositional encoding[5]. In our example, where $\kappa = \langle h_2, \tilde{h}\rangle \cdot k$, it must be split into $\langle h_2, \tilde{h}\rangle \cdot k$ and $\langle h_3, \tilde{h}\rangle \cdot k$, with the new h_3 included in $\tilde{h}$ and propagated to $\langle h_1, \tilde{h}\rangle \cdot k : P$. To resolve this, we outline two possible approaches:

- **Extending roll-π with nested tag splitting**: Introduce a variant where tags can be split at multiple levels, e.g. producing tags like $\langle h, \tilde{h}\rangle \cdot \langle \ell, \tilde{\ell}\rangle \cdot k$. This allows finer-grained control over tag structures but requires proving its encodability in standard **roll-π**.
- **Allowing choice at the configuration level**: Permit choices not only in processes but also in configurations, e.g. $\kappa_1 : L \oplus_\gamma \kappa_2 : R$. With a suitable structural congruence rule, $k : L \oplus_\gamma R \equiv \langle h_1, \tilde{h}\rangle \cdot k : L \oplus_\gamma \langle h_2, \tilde{h}\rangle \cdot k : R$ follows. However, ensuring a well-behaved interaction between this rule and (E.TagP) remains a challenge.

Conjectured Results. We investigate the relation between **roll-π!⊕** and **roll-π!**, conjecturing that they form a *weak barbed bisimulation*, regardless of how the tag issue is addressed. This follows from the need for multiple transitions to make a choice: one to select it and others to consume the remaining options. Combined with the strong barbed bisimulation between **roll-π!** and **roll-π** discussed in § 2, we aim to demonstrate that a weak barbed bisimulation exists between **roll-π!⊕** and **roll-π**.

4 Conclusion

We presented ongoing work on integrating choice and replication into **roll-π**, constructs more straightforwardly encoded in **HOπ**. Our early results indicate that encoding choice introduces additional challenges due to the complexities of handling process tags.

To accommodate reversibility, our encodings of replication and choice differ from standard approaches. While replication is typically handled through structural congruence [8,13], in our framework, it requires an explicit reduction. This may affect the termination of processes, and therefore, termination should be considered up to replication. For choice, we adopt an internal model where the process decides locally between branches. In practice, however, external choice – where the decision depends on the interactions available in the surrounding context – may be preferred [8,9,13]. Nevertheless, as demonstrated in our example, reversibility provides an effective rollback mechanism for internal choices, facilitating the derivation of external choices.

Future work will include the integration of faults and failure detection in the calculus, for example, by introducing a rule that reduces any process to a failure

[5] While this encoding likely preserves bisimilarity, it complicates characterising related configurations, as it requires renaming keys in multiple places, making proofs unnecessarily complex.

detector P_{fd}. This aligns with our goal of developing a multi-party session-typed calculus with transparent fault-tolerance based on **roll**-π.

Acknowledgments. This research was partially funded by EPSRC EP/T006544/2, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, Horizon EU TaRDIS 101093006.

References

1. Barbanera, F., Lanese, I., de'Liguoro, U.: A theory of retractable and speculative contracts. Science of Computer Programming **167**, 25–50 (2018). https://doi.org/https://doi.org/10.1016/j.scico.2018.06.005, https://www.sciencedirect.com/science/article/pii/S0167642318302636
2. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Concurrent reversible sessions. In: Meyer, R., Nestmann, U. (eds.) 28th International Conference on Concurrency Theory (CONCUR 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 85, pp. 30:1–30:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2017). https://doi.org/10.4230/LIPIcs.CONCUR.2017.30, http://drops.dagstuhl.de/opus/volltexte/2017/7787, ISSN: 1868-8969
3. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. ACM SIGPLAN Notices **43**(1), 273–284 (2008). https://doi.org/10.1145/1328897.1328472, https://dl.acm.org/doi/10.1145/1328897.1328472
4. Lagaillardie, N., Neykova, R., Yoshida, N.: Stay safe under panic: Affine rust programming with multiparty session types. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 4:1–4:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.4, https://drops.dagstuhl.de/opus/volltexte/2022/16232, ISSN: 1868-8969
5. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.B.: Controlling reversibility in higher-order pi. In: Katoen, J.P., König, B. (eds.) CONCUR 2011 – Concurrency Theory. pp. 297–311. Springer Berlin Heidelberg (2011), https://link.springer.com/chapter/10.1007/978-3-642-23217-6_20
6. Mezzina, C.A.: Reversing execution in higher-order pi, https://tel.archives-ouvertes.fr/tel-00683964
7. Mezzina, C.A., Tiezzi, F., Yoshida, N.: Checkpoint-based rollback recovery in session programming. Logical Methods in Computer Science **Volume 21, Issue 1** (Jan 2025). https://doi.org/10.46298/lmcs-21(1:2)2025, https://lmcs.episciences.org/15049
8. Milner, R.R.: Communicating and mobile systems : the pi-calculus. Cambridge [England] ; New York : Cambridge University Press (1999), http://archive.org/details/communicatingmob00robi
9. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, II. Information and Computation **100**(1), 41–77 (1992). https://doi.org/10.1016/0890-5401(92)90009-5, http://www.sciencedirect.com/science/article/pii/0890540192900095
10. Perumalla, K.S., Park, A.J.: Reverse computation for rollback-based fault tolerance in large parallel systems. Cluster Computing **17**(2), 303–313 (Jun 2014). https://doi.org/10.1007/s10586-013-0277-4, https://doi.org/10.1007/s10586-013-0277-4

11. Peters, K., Nestmann, U., Wagner, C.: Fault-tolerant multiparty session types. In: Mousavi, M.R., Philippou, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. pp. 93–113. Lecture Notes in Computer Science, Springer International Publishing (2022). https://doi.org/10.1007/978-3-031-08679-3_7
12. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2011). https://doi.org/10.1017/CBO9780511777110
13. Sangiorgi, D., Walker, D.: The Pi-Calculus - a theory of mobile processes. Cambridge University Press (2001)
14. Vassor, M., Stefani, J.B.: Checkpoint/rollback vs causally-consistent reversibility. In: Kari, J., Ulidowski, I. (eds.) Reversible Computation. pp. 286–303. Lecture Notes in Computer Science, Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-99498-7_20
15. Vidal, G.: From Reversible Computation to Checkpoint-Based Rollback Recovery for Message-Passing Concurrent Programs. In: Cámara, J., Jongmans, S.S. (eds.) Formal Aspects of Component Software. pp. 103–123. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-52183-6_6