Rollback Recovery in Session-based Programming

Claudio Antares Mezzina\(^1\), Francesco Tiezzi\(^2\), and Nobuko Yoshida\(^3\)

\(^1\) University of Leicester, United Kingdom  
cam7@leicester.ac.uk  
\(^2\) University of Camerino, Italy  
francesco.tiezzi@unicam.it  
\(^3\) Imperial College London, United Kingdom  
n.yoshida@imperial.ac.uk

Abstract. To react to unforeseen circumstances or amend abnormal situations (e.g., runtime errors) in communication-centric systems, programmers are in charge of “undoing” the interactions which led to an undesired state. To assist this task, session-based languages can be endowed with reversibility mechanisms. In this paper we propose a language enriched with programming facilities to commit session interactions, to roll back the computation to a previous commit point, and to abort the session. We prove that rollbacks in our language always bring the system to previous visited states and that a rollback cannot bring the system back to a point prior to the last commit. Programmers are relieved from the burden of ensuring that a rollback never restores a checkpoint imposed by a session participant different from the rollback requester. Such undesired situations are prevented at design-time (statically) by relying on a decidable compliance check at the type level, implemented using MAUDE. We show that the language satisfies error-freedom and progress of a session, demonstrate the practical effectiveness of the proposed approach by a means of a speculative execution scenario from the reversibility literature, and illustrate how the approach applies to both binary and multiparty sessions.

Keywords: Reversible Computing · Session Types · Recovery · Rollback · Checkpoints · MAUDE

1 Introduction

Reversible computing, whose origin dates back to the 70’s for circuits energy efficiency, is now attracting interest for its application to different fields: from modelling biological/chemical phenomena \([18]\) to simulation \([30]\), debugging \([12]\) and modelling fault-tolerant systems \([11][19]\). Our interest focusses on this latter application and, in particular, stems from the fact that reversibility can be used to rigorously model, implement and revisit programming abstractions for reliable software systems.

Recent works \([3][27][26][32]\) have studied the effect of reversibility in communication-centric scenarios, as a means to amend computations that led to an error by bringing back the system to a consistent previous state. In this setting, processes’ behaviours are strongly disciplined by their types, prescribing the actions they have to perform within a session. A session consists in a structured series of message exchanges, whose
flow can be controlled via conditional choices, branching and recursion. Correctness of communication is statically guaranteed by a framework based on a (session) type discipline [16]. Nonetheless, none of these works addresses systems in which the participants can explicitly abort the session, commit a computation and roll it back to a previous checkpoint. In this paper, we aim at filling this gap in programming reversible behaviours within communication-centric systems. We explain below the distinctive aspects of our checkpoint-based rollback recovery approach.

**Linguistic primitives for explicitly and practically programming reversible sessions.** We introduce three primitives to: (i) commit a session, preventing undoing the interactions performed so far along the session; (ii) roll back a session, restoring the last saved process checkpoints; (iii) abort a session, to discard the session, and hence all interactions already performed in it, thus allowing another session of the same protocol to start with possible different participants. Notice that most proposals in the literature (e.g., [1,2,3]) only consider an abstract view, as they focus on reversible contracts (i.e., types). Instead, we focus on programming primitives at process level, and use types for guaranteeing a safe and consistent system evolution.

**Asynchronous commits.** Our commit primitive does not require a session-wide synchronisation among all participants, as it is a local decision. However, its effect is on the whole session, as it affects the other session participants. This means that each participant can independently decide when to commit. Such flexibility comes at the cost of being error-prone, especially considering that the programmer has not only to deal with the usual forward executions, but also with the backward ones. Our type discipline allows for ruling out programs which may lead to these errors.

The key idea of our approach is that “a session participant executing a rollback action is interested in restoring the last checkpoint he/she has committed”. For the success of the rollback recovery it is irrelevant whether the ‘passive’ participants go back to their own last checkpoints. Instead, if the ‘active’ participant is unable to restore the last checkpoint he/she has created, because it has been replaced by a checkpoint imposed by another participant, the rollback recovery is considered unsatisfactory.

In our framework, programmers are relieved from the burden of ensuring the satisfaction of rollbacks, since undesired situations are prevented at design time (statically) by relying on a compliance check at the type level. To this aim, we introduce cherry-\(\pi\)-calculus (checkpoint-based rollback recovery \(\pi\)-calculus), a variant of the session-based \(\pi\)-calculus [38,17] enriched with rollback recovery primitives. We present first a binary version of the calculus, which is more convenient to demonstrate the essence of our rollback recovery approach, and then we extend it to multiparty sessions. A key difference with respect to the standard binary type discipline is the relaxation of the duality requirement. The types of two session participants are not required to be dual, but they will be compared with respect to a compliance relation (as in [2]), which also takes into account the effects of commit and rollback actions. Such relaxation also involves the requirements concerning selection and branching types, and those concerning branches of conditional choices. The cherry-\(\pi\) type system is used to infer types of session participants, then combined together for the compliance check.

Reversibility in cherry-\(\pi\) is controlled via two specific primitives: a rollback one telling when a reverse computation has to take place, and a commit one limiting the scope
of a potential reverse computation. This implies that the calculus is not fully reversible (i.e., backward computations are not always enabled), leading to have relaxed and different properties with respect to other reversible calculi, such as RCCS \cite{10}, Rπ \cite{9}, ρπ \cite{21,22} and ReSπ \cite{32}. We prove that cherry-pi satisfies the following properties: (i) a rollback always brings back the system to a previous visited state and (ii) it is not possible to bring the computation back to a point prior to the last checkpoint, which implies that our commits have a persistent effect. Concerning soundness properties, we prove that (a) our compliance check is decidable, and that compliance-checked cherry-pi specifications (b) never reduce to communication errors (e.g., a blocked communication where there is a receiver without the corresponding sender) and (c) never activate undesirable rollbacks (according to our notion of rollback recovery mentioned above). Property (b) resembles the type safety property of session-based calculi (see, e.g., \cite{38}), while property (c) is a new property specifically defined for cherry-pi. The technical development of property proofs turns out to be more intricate than that of standard properties of session-based calculi, due to the combined use of type and compliance checking.

To demonstrate feasibility and effectiveness of our rollback recovery approach, we have concretely implemented\footnote{Our MAUDE implementation of the compliance check is available at the cherry-pi’s website: \url{http://pros.unicam.it/tools/cherry-pi}} the compliance check using the MAUDE framework \cite{7}.

Outline. Sec. 2 illustrates the key idea of our rollback recovery approach; Sec. 3 introduces syntax and semantics of the cherry-pi calculus; Sec. 4 introduces typing and compliance checking; Sec. 5 presents the properties satisfied by cherry-pi; Sec. 6 shows an application of cherry-pi to a practical case study \cite{31,14}; Sec. 7 describes how the approach extends to multiparty sessions; Sec. 8 concludes the paper with related and future work. The Appendix reports the proofs of our results.

2 A reversible video on demand service example

We discuss the motivations underlying our work by introducing our running example, a Video on Demand (VOD) scenario. The key idea is that a rollback requester is satisfied only if her restored checkpoint was set by herself. In Fig. 1(a), a service (S) offers to a user (U) videos with two different quality levels, namely high definition (HD) and standard definition (SD). After the login, U sends her video request, and receives the corresponding price and metadata (actors, directors, description, etc.) from S. According to this information, U selects the video quality. Then, she receives, first, a short test video (to check the audio and video quality in her device) and, finally, the requested video. If the vision of the HD test video is not satisfactory, U can roll back to her last checkpoint to possibly change the video quality, instead in the SD case U can abort the session.

Let us now add commit actions as in the run shown in Fig. 1(b). After receiving the price, U commits, while S commits after the quality selection. In this scenario, however, if U activates the rollback, she is unable to go back to the checkpoint she set with her commit action because the actual effect of rollback is to restore the checkpoint set by the commit action performed by S.
In the scenario in Fig. 1(c), instead, S commits after sending the price to U. In this case, no matter who first performed the commit action, the rollback results to be satisfactory. Also if S commits later, the checkpoint of U remains unchanged, as U performed no other action between the two commit actions. This would not be the case if both U and S committed after the communication of the metadata, as in Fig. 1(d). If S commits before U, no rollback issue arises, but if U commits first it may happen that her internal decision is taken before that S commits. In this case, U would not be able to go back to the checkpoint set by herself, and she would be unable to change the video quality.

These undesired rollbacks are caused by bad choices of commit points. We propose a compliance check that identifies these situations at design time, avoiding unnecessary checks at runtime.

3 The cherry-pi calculus

In this section, we introduce cherry-pi, a calculus (extending that in [38]) devised for studying sessions equipped with our checkpoint-based rollback recovery mechanism.

3.1 Syntax

The syntax of the cherry-pi calculus relies on the following base sets: shared channels (ranged over by \( a \)), used to initiate sessions; session channels (ranged over by \( s \)), consisting of pairs of endpoints (ranged over by \( s, \bar{s} \)) used by the two parties to interact within an established session; labels (ranged over by \( l \)), used to select and offer branching choices; values (ranged over by \( v \)), including booleans, integers and strings (whose sorts, ranged over by \( S \), are \( \text{bool} \), \( \text{int} \) and \( \text{str} \), respectively), which are exchanged within a
Fig. 2. cherry-pi syntax.

session; variables (ranged over by \( x, y, z \)), storing values and session endpoints; process variables (ranged over by \( X \)), used for recursion.

*cherry-pi* specifications, called *collaborations* and ranged over by \( C \), are given by the grammar in Fig. 2. The key ingredient of the calculus is the set of actions for controlling the session rollback. Actions commit, roll and abort are used, respectively, to commit a session (producing a checkpoint for each session participant), to trigger the session rollback (restoring the last committed checkpoints) or to abort the whole session. We discuss below the other constructs of the calculus, which are those typically used for session-based programming [15].

A *cherry-pi* collaboration is a collection of *session initiators*, i.e. terms ready to initiate sessions by synchronising on shared channels. A synchronisation of two initiators \( a(x).P \) and \( a(y).Q \) causes the generation of a fresh session channel, whose endpoints replace variables \( x \) and \( y \) in order to be used by the triggered processes \( P \) and \( Q \), respectively, for later communications. No subordinate sessions can be initiated within a running session.

When a session is started, each participant executes a *process*. Processes are built up from the process 0, which does nothing, and from basic actions by means of action prefix _|_, conditional choice if e then _|_ else _, and recursion \( \mu X. \_ \). Actions \( x!()P \) and \( y?(z : S)P \) denote output and input via session endpoints replacing \( x \) and \( y \), respectively. These communication primitives realise the standard synchronous message passing, where messages result from the evaluation of *expressions*, which are defined by means of standard operators on boolean, integer and string values. Variables that are arguments of input actions are (statically) typed by sorts. There is no need for statically typing the variables occurring as arguments of session initiating actions, as they are always replaced by session endpoints. Notice that in *cherry-pi* the exchanged values cannot be endpoints, meaning that session delegation (i.e., channel-passing) is not considered. Actions \( x<1 \) and \( x \Rightarrow [l_1 : P_1, \ldots, l_n : P_n] \) denote selection and branching (where \( l_1, \ldots, l_n \) are pairwise distinct).

**Example 1.** Let us consider the VOD example informally introduced in Sec. 2. The scenario described in Fig. 1(a) with commit actions placed as in Fig. 1(b) is rendered in *cherry-pi* as

\[
C_{US} = \overline{\text{login}(x)}.P_U \mid \text{login}(y).P_S
\]
where:

\[
P_0 = \pi_0(\pi_1) . \pi_2
\]

\[
P_s = \pi_s(\pi_t) . \pi_u
\]

Notice that expressions used for decisions and computations are abstracted by functions \( f_\sigma(\cdot) \).

Considering the placement of commit actions depicted in Fig. 1(c), the cherry-pi specification of the service’s process becomes:

\[
y!(\pi_1) . \pi_2
\]

Finally, considering the placement of commit actions depicted in Fig. 1(d), the cherry-pi specification of the user’s process becomes:

\[
x!(\pi_1) . \pi_2
\]

### 3.2 Semantics

The operational semantics of cherry-pi is defined for runtime terms, generated by the extended syntax of the calculus in Fig. 3 (where the new constructs are highlighted by a grey background). We use \( k \) to denote generic session endpoints \( s \) or \( \bar{s} \); and \( r \) to denote session identifiers, i.e., session endpoints and variables together. We refer as initial collaborations to those runtime terms that can be also generated by the grammar in Fig. 2.

At collaboration level, two constructs are introduced: \( (\pi_s : C_1) C_2 \) represents a session along the channel \( s \) with associated starting checkpoint \( C_1 \) (corresponding to the collaboration that has initialised the session) and code \( C_2 \); construct \( \pi_s \) \( P_1 \) \( P_2 \) represents a log storing the checkpoint \( P_1 \) associated to the code \( P_2 \). At process level, the only difference is that session identifiers \( r \) are used as first argument of communicating actions. We extend the standard notion of binders to take into account \( (\pi_s : C_1) C_2 \), which binds session endpoints \( s \) and \( \bar{s} \) in \( C_2 \) (in this respect, it acts similarly to the restriction of \( \pi \)-calculus, but its scope cannot be extended in order to avoid involving...
processes that do not belong to the session in the rollback effect). The derived notions of bound and free names (where names stand for variables, process variables and session endpoints), alpha-equivalence, and substitution are standard and we assume that bound names are pairwise distinct. The semantics of the calculus is defined for closed terms, i.e. terms without free variables and process variables.

Not all processes allowed by the extended syntax corresponds to meaningful collaborations. In a general term the processes stored in logs may not be consistent with the computation that has taken place. We get rid of such malformed terms, as we will only consider those runtime terms, called reachable collaborations, obtained by means of reductions from initial collaborations.

In cherry-pi, processes can perform, besides usual forward computations, also backward computations that restore the last committed checkpoints, thus undoing the effect of forward computations. The operational semantics is given in terms of a standard structural congruence [17], written \( \equiv \), and a reduction relation, written \( \rightarrow \), given as the union of the corresponding forward reduction relation \( \rightarrow' \) and backward reduction relations \( \rightarrow'' \). The definition of the relation \( \rightarrow' \) over closed collaborations relies on an auxiliary labelled relation \( \ell \) over processes that specifies the actions that processes can initially perform and the continuation process obtained after each such action. The notation of the above relations is summarised in Fig. 4. Given a reduction relation \( \mathcal{R} \), we will indicate with \( \mathcal{R}^+ \) and \( \mathcal{R}^* \) respectively the transitive and the reflexive-transitive closure of \( \mathcal{R} \).

The operational rules defining the auxiliary labelled relation are reported in Fig. 5 where action label \( \ell \) stands for either \( k! \langle \psi \rangle, k? (x), k \leftarrow l, k \rightarrow l, \text{cmt}, \text{roll}, \text{abt}, \) or \( \tau \). The meaning of these rules is straightforward, as they just produce as labels the actions currently enabled in the process. In doing that, expressions of sending actions and conditional choices are evaluated (the auxiliary function \( e \downarrow v \) says that closed expression \( e \) evaluates to value \( v \)).

The operational rules defining the reduction relation \( \rightarrow \) are reported in Fig. 6. We comment on salient points. Once a session is created, its initiating collaboration is
Example 2. Consider the first cherry-pi specification of the VOD scenario given in EX.\[\text{I}\] in the initial state $C_{\text{US}}$ of the collaboration, $\cup$ and $\subseteq$ can synchronise in order to initialise the session, thus evolving to

$$C_{\text{US}}^3 = (\nu s : C_{\text{US}}) (\langle P_\cup[s/x] \rangle \bullet P_\cup[s/x] | \langle P_\subseteq[s/y] \rangle \bullet P_\subseteq[s/y])$$

stored in the session construct (rule [F-CON]). Communication, branching selection and internal conditional choices proceed as usual, without affecting logs (rules [F-COM], [F-LAB] and [F-IF]). A commit action updates the checkpoint of a session, by replacing the processes stored in the logs of the two involved parties (rule [F-CMT]). Conversely, a rollback action restores the processes in the two logs (rule [B-RLL]). The abort action, instead, restores the collaboration stored in the session construct formed by the two initiators that have started the session (rule [B-ABT]). The other rules simply extend the standard parallel, restriction, structural congruence rules to forward and backward relations.

---

Fig. 6. cherry-pi semantics: forward and backward reduction relations.
Let us consider now a possible run of the session. After three reduction steps, $U$ executes the commit action, obtaining the following runtime term:

$$C_2^U = (\nu S : C_{US}) \langle \langle P'_{U} \rangle \triangleright P'_{S} \mid \langle P'_{S} \rangle \triangleright P'_{S} \rangle$$

$$P'_{U} = s?(x_{\text{meta}} : \text{str}). \text{if} \ (f_{\text{eval}}(v_{\text{price}}, x_{\text{meta}})) \ \text{then} \ldots$$

$$P'_{S} = s!(f_{\text{meta}}(v_{\text{req}})). y \triangleright \{ \ldots \}$$

After four further reduction steps, $U$ chooses the HD video quality and $S$ commits as well; the resulting runtime collaboration is as follows:

$$C_3^U = (\nu S : C_{US}) \langle \langle P''_{U} \rangle \triangleright P''_{S} \mid \langle P''_{S} \rangle \triangleright P''_{S} \rangle$$

$$P''_{U} = s?(x_{\text{testHD}} : \text{str}). \text{if} \ (f_{\text{HD}}(x_{\text{testHD}})) \ \text{then} \ s?(x_{\text{videoHD}} : \text{str}).0 \ \text{else roll}$$

$$P''_{S} = s!(f_{\text{testHD}}(v_{\text{req}})). s!(f_{\text{videoHD}}(v_{\text{req}})). 0$$

In the next reductions, $U$ evaluates the test video and decides to revert the session execution, resulting in

$$C_4^U = (\nu S : C_{US}) \langle \langle P'''_{U} \rangle \triangleright \text{roll} \mid \langle P'''_{S} \rangle \triangleright s!(f_{\text{videoHD}}(v_{\text{req}})). 0 \rangle$$

The execution of the roll action restores the checkpoints $P''_{U}$ and $P''_{S}$, that is $C_4^U \rightsquigarrow C_3^U$. After the rollback, $U$ is not able to change the video quality as her own commit point would have permitted; in fact, it holds $C_4^U \not\rightsquigarrow C_2^U$.

### 4 Rollback safety

The operational semantics of cherry-pi provides a description of the functioning of the primitives for programming the checkpoint-based rollback recovery in a session-based language. However, as shown in Ex. 2, it does not guarantee high-level properties about the safe execution of the rollback. To prevent such undesired rollbacks, we propose the use of compliance checking, to be performed at design time before actually executing the system. This check is not done on the full system specification, but only at the level of session types.

#### 4.1 Session types and typing

The syntax of the cherry-pi session types is defined in Fig. 7. The type $![S].T$ represents the behaviour of first outputting a value of sort $S$, then performing the actions prescribed by type $T$. The type $?[S].T$ is the dual one, where a value is received instead of sent. Type $\text{end}$ represents inaction. Type $\llbracket l \rrbracket.T$ represents the behaviour that selects the label $l$ and then behaves as $T$. Type $\ggbracket_{1:T_1, \ldots, t_n : T_n}$ describes a branching behaviour: it waits with $n$ options, and behaves as type $T_i$ if the $i$-th label is selected (external choice). Type $T_1 \oplus T_2$ behaves as either $T_1$ or $T_2$ (internal choice). Type $\mu t.T$ represents a recursive behaviour that starts by doing $T$ and, when variable $t$ is encountered, recurs to $T$ again. Type $\text{cmt}.T$ represents a commit action followed by the actions prescribed by type $T$. Finally, types $\text{roll}$ and $\text{abt}$ represent rollback and abort actions.
The `cherry-pi` type system does not perform compliance checks, but it just aims at inferring the types of collaboration participants, which will be then checked together according to the compliance relation. Typing judgements are of the form $\mathcal{C} \triangleright A$, where $A$, called type associations, is a set of session type associations of the form $\hat{a} : T$, where $\hat{a}$ stands for either $\bar{a}$ or $a$. Intuitively, the judgement $\mathcal{C} \triangleright A$ indicates that from the collaboration $\mathcal{C}$ the type associations in $A$ are inferred. The definition of the type system for these judgments relies on auxiliary typing judgements for processes, of the form $\Theta ; \Gamma \vdash P \trianglerightin \Delta$, where $\Theta$, $\Gamma$ and $\Delta$, called basis, sorting and typing respectively, are finite partial maps from process variables to type variables, from variables to sorts, and from variables to types, respectively. Updates of basis and sorting are denoted, respectively, by $\Theta \bar{\ast} \Delta$ and $\Gamma \bar{\ast} S$, where $\Delta$ stands for either $\bar{a}$ or $a$. Intuitively, the judgement $\Theta ; \Gamma \vdash P \trianglerightin \Delta$ stands for "under the environment $\Theta ; \Gamma$, process $P$ has typing $\Delta"$. In its own turn, the typing of processes relies on auxiliary (standard) judgments for expressions, of the form $\Gamma \vdash e \triangleright S$. The axioms and rules defining the typing system for `cherry-pi` collaborations, processes and expressions are given in Fig. 8 and 9 and 10 respectively. The type system is defined only for initial collaborations, i.e. for terms generated by the grammar in Fig. 2. Other runtime collaborations are not considered here, as no check will be performed at runtime. We comment on salient points. Typing rules at collaboration level simply collect the type associations of session initiators in the collaboration. Rules at process level instead determine the session type corresponding to each process, by mapping each process operator to the corresponding type operator. Data and expression used in communication actions are abstracted as sorts, and a conditional choice is rendered as an internal non-deterministic choice. Typing rules for expressions are standard.

4.2 Compliance checking

To check compliance between pairs of session parties, we consider type configurations of the form $(T, T') : \langle T_1 \rangle \cdot T_2 || \langle T_3 \rangle \cdot T_4$, consisting in a pair $(T, T')$ of session types,
corresponding to the types of the parties at the initiation of the session, and in the parallel composition of two pairs \((\tilde{T}_a, T)\) the type of the party’s checkpoint. We use \(\tilde{T}\) to denote either a type \(T\), representing a checkpoint committed by the party, or \(\bar{T}_c\), representing a checkpoint imposed by the other party of the session. The semantics of type configurations, necessary for the definition of the compliance relation, is given in Fig. 11, where label \(\lambda\) stands for either \(![S]\), \?[S]\), \(<\tilde{l}\), \(\tilde{r}\), \(\text{cmt}\), \(\text{roll}\), or \(\text{abt}\). We comment on the relevant rules. In case of a commit action, the checkpoints of both parties are updated, but the one of the passive party (i.e., the party that has not performed the commit action) is marked as ‘imposed’ (rule [TS-CMT]) . However, if the passive party did not perform any action from its current checkpoint, this checkpoint is not overwritten by the active party (rule [TS-CMT2]), as discussed in Sec. 2 (Fig. 1(c)). In case of a roll action (rule [TS-RLL1]), the reduction step is performed only if the active party (i.e., the party that has performed the rollback action) has a non-imposed checkpoint; in all other situations the configuration cannot proceed with the rollback. Finally, in case of abort (rule [TS-ABT1]), the configuration goes back to the initial state; this allows the type computation to proceed, in order to not affect the compliance check between the two parties.
Theorem 1. Let $T_1$ and $T_2$ be two session types, checking if $T_1 -\parallel T_2$ holds is decidable.
Definition 2 (Rollback safety). Let \( C \) be an initial collaboration, then \( C \) is rollback safe (shortened r-safe) if \( C \rightarrow A \) and for all pairs \( \bar{a} : T_1 \) and \( a : T_2 \) in \( A \) we have \( T_1 \setminus T_2 \).

Example 3. Let us consider again the VOD example. As expected, the first cherry-pi collaboration defined in Ex. 1 corresponding to the scenario described in Fig. 1(b), is not rollback safe, because the types of the two parties are not compliant. Indeed, the session types \( T_u \) and \( T_\delta \) associated by the type system to the user and the service processes, respectively, are as follows:

\[
T_u = ![\text{str}].?[\text{int}].\text{cm}.?[\text{str}].(\llcorner [\text{HD}].?[\text{str}].(?[\text{str}].\text{end} \oplus \text{roll})
\oplus \llcorner [\text{SD}].?[\text{str}].(?[\text{str}].\text{end} \oplus \text{abt}))
\]

\[
T_\delta = ![\text{str}].?[\text{int}].?[\text{str}]. \triangleright ![\text{str}] : \text{cms}.![\text{str}].![\text{str}].\text{end},
\]
\[
l_\text{SD} : ![\text{str}].?[\text{str}].?[\text{str}].\text{end}
\]

Thus, the resulting initial configuration is \((T_u, T_\delta) : \langle T_u \rangle \owns T_u \parallel \langle T_\delta \rangle \owns T_\delta\), which can evolve to the configuration \((T_u, T_\delta) : \langle T \rangle \owns \text{roll} \parallel \langle U \rangle \owns ![\text{str}].\text{end}\), with \( T = ![\text{str}].?[\text{str}].\text{end} \oplus \text{roll} \) and \( U = ![\text{str}].?[\text{str}].\text{end}\). This configuration cannot evolve and is not in a final state (in fact, types \( \text{roll} \) and \( ![\text{str}].\text{end} \) are different from \text{end}), meaning that \( T_u \) and \( T_\delta \) are not compliant.

In the scenario described in Fig. 1(c), instead, the type of the server process is as follows:

\[
T_\delta' = ![\text{str}].?[\text{int}].\text{cm}.?[\text{str}]. \triangleright ![\text{str}] : ![\text{str}].![\text{str}].\text{end},
\]
\[
l_\text{SD} : ![\text{str}].?[\text{str}].?[\text{str}].\text{end}
\]

and we have \( T_u \not\owns T_\delta' \).

Finally, the types of the processes depicted in Fig. 1(d) are:

\[
T_u' = ![\text{str}].?[\text{int}].?[\text{str}].\text{cm} (\llcorner [\text{HD}]. \ldots \oplus \llcorner [\text{SD}]. \ldots )
\]

\[
T_\delta'' = ![\text{str}].?[\text{int}].?[\text{str}].\text{cm} \triangleright ![\text{str}] : ![\text{str}].![\text{str}].\text{end},
\]
\[
l_\text{SD} : ![\text{str}].?[\text{str}].?[\text{str}].\text{end}
\]

and we have \( T_u' \not\owns T_\delta'' \). Indeed, the corresponding initial configuration can evolve to the configuration \((T_u', T_\delta'') : \langle [\text{HD}] \ldots \rangle \owns \text{roll} \parallel \langle ![\text{HD}] : [\ldots] \parallel ![\text{str}].\text{end}\rangle\), which again is not a final state.

4.3 MAUDE implementation

To practically enable the compliance checking at the basis of our notion of rollback safety, we have implemented the semantics of type configurations in Fig. 1 within the MAUDE framework [7]. MAUDE provides an instantiation of rewriting logic [24] and it has been used to implement the semantics of a variety of formal languages [25].

The syntax of cherry-pi types and type configurations is specified by defining algebraic data types, while transitions and reductions are rendered as rewrite and, hence, inference rules are given in terms of (conditional) rewrite rules. Since MAUDE
specifications are executable, we have obtained in this way an interpreter for cherry-pi type configurations, which permits to explore the reductions arising from the initial configuration of two given session types.

Our implementation consists of two MAUDE modules:

- **CHERRY-TYPES-SYNTAX** provides the definition of the sorts that characterise the syntax of cherry-pi types, such as session types, selection/branching labels, type variables and type configurations. In particular, basic terms of session types are rendered as constant operations on the sort Type; e.g., the roll type is defined as

  \[ \text{op roll : } \rightarrow \text{Type}. \]

  The other syntactic operators are instead defined as operations with one or more input arguments; e.g., the output type takes as input a Sort (i.e., bool, int or str) and a continuation type:

  \[ \text{op \_\_ : Sort Type } \rightarrow \text{Type [frozen prec 25].} \]

  To prevent undesired rewrites inside operator arguments, following the approach in [35], we declared these operations as **frozen**. The **prec** attribute has been used to define the precedence among operators.

- **CHERRY-TYPES-SEMANTICS** provides rewrite rules, and additional operators and equations, to define the cherry-pi type semantics. For example, the operational rule \([\text{TS-Snd}]\) of the output type, i.e. \(\text{!}\{S\}.T \xrightarrow{\text{!}\{S\}} T\), is rendered as the following rewrite rule:

  \[ \text{rl [TS-Snd] : \!\{S\}.T \Rightarrow \{\!\{S\}\}T.} \]

  The correspondence between the operational rule and the rewrite one is one-to-one; the only peculiarity is the fact that, since rewrites have no labels, we have made the transition label part of the resulting term. Reduction rules for type configurations are instead rendered in terms of (conditional) rewrite rules with rewrites in their conditions. For example, the \([\text{TS-Com}]\) rule is rendered as:

  \[ \text{rl [TS-Com] : init(T,T') CT1 > T1 || CT2 > T2 } \Rightarrow \text{init(T,T') CT1 > T1' || CT2 > T2' if T1 \Rightarrow \{\!\{S\}\}T1' \land T2 \Rightarrow \{?\{S\}\}T2'.} \]

  The compliance check between two session types can be then conveniently realised on top of the implementation described above by resorting to the MAUDE command **search**. This permits indeed to explore the state space of configurations reachable from an initial configuration. Specifically, the compliance check between types \(T1\) and \(T2\) is rendered as follows:

  \[ \text{search \ init(T1,T2) ckp(T1) > T1 || ckp(T2) > T2 \ n=1; \ init(T:Type,T':Type) CT1:KnpType > T1':Type || CT2:KnpType > T2':Type such that T1' !=/ end or T2' !=/ end.} \]

  This command searches for all terminal states \(\Rightarrow !\), i.e. states that cannot be rewritten any more (see \(-\rightarrow\) in Def. [1]), and checks if at least one of the two session types in the
corresponding configurations (T_1' and T_2') is different from the end type. Thus, if this search has no solution, T_1 and T_2 are compliant; otherwise, they are not compliant and a violating configuration is returned.

Example 4. Let us consider the cherry-pi types defined in Ex. 3 for the scenario described in Fig. 1(b). In our MAUDE implementation of the type syntax, the session types T_U and T_S, and the corresponding initial type configuration, are rendered as follows:

\[
\begin{align*}
\text{eq } & T\text{user} = ![\text{str}]. ?[\text{int}]. \text{cmt}. ?[\text{str}]. \\
& (\text{sel} ['hd']. ?[\text{str}]. ((?[\text{str}]. \text{end}) (+) \text{roll})) \\
& (+) (\text{sel} ['sd']. ?[\text{str}]. ((?[\text{str}]. \text{end}) (+) \text{abt})) .
\end{align*}
\]

\[
\begin{align*}
\text{eq } & T\text{service} = ?[\text{str}]. ![\text{int}]. ![\text{str}]. \\
& \text{brn}[\text{brnEl}('hd, \text{cmt}. ![\text{str}]. ![\text{str}]. \text{end}); \\
& \text{brnEl}('sd, \text{cmt}. ![\text{str}]. ![\text{str}]. \text{end})] .
\end{align*}
\]

\[
\begin{align*}
\text{eq } & \text{InitConfig} = \text{init}(T\text{user}, T\text{service}) \\
& \text{ckp}(\text{Tuser}) > T\text{user} || \text{ckp}(\text{Tservice}) > T\text{service} .
\end{align*}
\]

where (+) represents the internal choice operator, sel the selection operator, brn the branching operator, brnEl an option offered in a branching, and ckp a non-imposed checkpoint. The compliance between the two session types can be then checked by first loading the two modules of our MAUDE implementation:

```
./maude.darwin64 cherry-pi_type_semantics.maude
```

and then executing the following command:

```
search InitConfig =! init(T:Type,T':Type) CT1:CkpType > T1:Type || CT2:CkpType > T2:Type \\
such that T1 =/= end or T2 =/= end .
```

This search command returns the following solution:

```
CT1 --> ickp(![str]. !([str]. end) (+) roll))
T1 --> roll
CT2 --> ckp(![str]. ![str]. end)
T2 --> ![str]. end
```

As already explained in Ex. 3, the two types are not compliant. Indeed, the configuration above is final, because T_1 cannot perform the rollback, since its checkpoint is imposed (ickp), and T_2 cannot perform the communication action, since its partner T_1 is not ready to receive. Moreover, roll and ![str]. end are clearly different from end.

The scenario in Fig. 1(c) is rendered by means of the following implementation of the service type:

\[
\begin{align*}
\text{eq } & T\text{service}' = ![\text{str}]. ![\text{int}]. \text{cmt}. ![\text{str}]. \\
& \text{brn}[\text{brnEl}('hd, ![\text{str}]. ![\text{str}]. \text{end}); \\
& \text{brnEl}('sd, ![\text{str}]. ![\text{str}]. \text{end})] .
\end{align*}
\]

In this case, as expected, the search command returns:

```
No solution.
```

5 We have used the version 2.7.1 of MAUDE for Mac OS X.
meaning that the types are compliant.

Finally, the search command applied to the type configuration related to the scenario depicted in Fig. 1(d) returns a solution, meaning that the user and service types are not compliant.

5 Properties of \textit{cherry-pi}

We present in this section the results regarding the properties enjoyed by \textit{cherry-pi}. Their proofs are reported in Appendix A. The statement of some properties exploits labelled transitions that permit to easily distinguish the execution of commit and rollback actions from the other ones. To this aim, rules [F-CMT], [B-RLL] and [B-ABT] can be slightly modified by labelling the transition in their conclusions by \texttt{cmt}, \texttt{roll} and \texttt{abt}, respectively, and rules [F/B-PAR], [F/B-RES] and [F/B-STR] change accordingly in order to propagate the labelled transitions.

5.1 Rollback properties

We show some properties related to the reversible behaviour of \textit{cherry-pi} and in particular related to the interplay between rollback and commit primitives.

The following theorem states that any reachable collaboration is also a forward only reachable collaboration. This means that all the states a collaboration reaches via backward reductions are states that we can reach from the initial configuration with just forward reductions. This assures us that if the system goes back it will reach previous visited states.

\textbf{Theorem 2.} Let $C_0$ be an initial collaboration. If $C_0 \rightarrow^* C_1$ then $C_0 \rightarrow^* C_1$.

We now show a variant of the so-called Loop Lemma [10]. In a fully reversible calculus this lemma states that each computational step, either forward or backward, can be undone. Since reversibility in \textit{cherry-pi} is controlled, we have to state that if a reversible step is possible (e.g., a rollback is enabled) then the effects of the rollback can be undone. Formally we have:

\textbf{Lemma 1 (Safe rollback).} Let $C_1$ and $C_2$ be reachable collaborations. If $C_1 \rightsquigarrow C_2$ then $C_1 \rightarrow^* C_1$.

The following result states that a rollback always brings the system to the last taken checkpoint.

\textbf{Lemma 2 (Determinism).} Let $C$ be a reachable collaboration. If $C \rightsquigarrow C'$ and $C \rightsquigarrow C''$ then $C' \equiv C''$.

The last rollback property states that a \textit{cherry-pi} collaboration cannot go back to a state prior to the execution of a commit action, that is commits have a persistent effect.

\textbf{Theorem 3 (Undoability).} Let $C$ be a reachable collaboration. If $C \rightarrow^* C'$ then there exists no $C''$ such that $C' \rightarrow^* \rightsquigarrow C''$ and $C'' \rightarrow^+ C$. 

The second group of properties concerns soundness guarantees. The definition of these properties requires to formally characterise the errors that may occur on the execution of an unsound collaboration. We rely on error reduction (as in [6]) rather than on the usual static characterisation of errors (as, e.g., in [33]), since rollback errors cannot be easily detected statically. In particular, we extend the syntax of cherry-pi collaborations with the roll_error and com_error terms, denoting respectively collaborations in rollback and communication error states:

\[
C ::= a(x).P \mid a(x).P \mid C_1 \mid C_2 \mid (\nu s : C_1) \ C_2 \\
\mid \langle P_1 \rangle \mid P_2 \mid \text{roll_error} \mid \text{com_error}
\]

where \( P \) denotes either a checkpoint committed by the party or a checkpoint \( P \) imposed by the other party of the session. The semantics of cherry-pi is extended as well by the (error reduction) rules in Fig. 12 where [E-CMT₁] and [E-CMT₂] replace [E-CMT], and [E-RLL₁] replaces [E-RLL], and \( \hat{=} \) is used in the checkpoints of the other rules. The error semantics does not affect the normal behaviour of cherry-pi specifications, but it is crucial for stating our soundness theorems. Its definition is based on the notion of \( \text{barb} \) predicate: \( P \Downarrow \mu \) holds if there exists \( P' \) such that \( P \Rightarrow P' \) and \( P' \) can perform an action \( \mu \), where \( \mu \) stands for \( k? \), \( k! \), \( k \leftarrow \), \( k \rightarrow \) (i.e., input, output, select and branching action along session channel \( k \)); \( \Rightarrow \) is the reflexive and transitive closure of \( \Rightarrow \). The meaning of the error semantics rules is as follows. A communication error takes place in a collaboration when a session participant is willing to perform an output but the other participant is not ready to perform the corresponding input (rule [E-CM₁]) or viceversa (rule [E-CM₂]), or one participant is willing to perform a selection...
but the corresponding branching is not available on the other side (rule [E-LAB1]) or vice versa (rule [E-LAB2]). Instead, a rollback error takes place in a collaboration when a participant is willing to perform a rollback action but its checkpoint has been imposed by the other participant ([E-RLL1]). To enable this error check, the rules for commit and rollback have been modified to keep track of imposed overwriting of checkpoints. This information is not relevant for the runtime execution of processes, but it is necessary for characterising the rollback errors that our type-based approach prevents.

Besides defining the error semantics, we also need to define erroneous collaborations, based on the following notion of context: $C ::= \{ \} \mid C \parallel C \mid (\nu s : C) C$.

**Definition 3** (Erroneous collaborations). A collaboration $C$ is communication (resp. rollback) erroneous if $C = C[com_error]$ (resp. $C = C[roll_error]$).

The key soundness results follow: a rollback safe collaboration never reduces to either a rollback erroneous collaboration (Theorem 4) or a communication erroneous collaboration (Theorem 5).

**Theorem 4** (Rollback soundness). If $C$ is a $r$-safe collaboration, then we have that $C \not\rightarrow^* C[roll_error]$.

**Theorem 5** (Communcation soundness). If $C$ is a $r$-safe collaboration, then we have that $C \not\rightarrow^* C[com_error]$.

We conclude with a property concerning the progress of cherry-pi sessions: given a rollback safe collaboration that can initiate a session, each collaboration reachable from it either is able to progress on the session with a forward/backward reduction step or has correctly reached the end of the session. This result straightforwardly follows from Theorems 4 and 5, and from the fact that we consider binary sessions without delegation and subordinate sessions.

**Theorem 6** (Session progress). Let $C = (\bar{a}(x_1).P_1 \mid a(x_2).P_2)$ be a $r$-safe collaboration. If $C \not\rightarrow^* C'$ then either $C' \not\rightarrow C''$ for some $C''$ or $C' = (\nu s : C)(\langle Q_1 \cdot 0 \rangle \parallel \langle Q_2 \cdot 0 \rangle)$ for some $Q_1$ and $Q_2$.

6 cherry-pi at work on a speculative parallelism scenario

To shed light on the practical effectiveness of cherry-pi and the related notion of rollback safety, we consider in this section a simple, yet realistic, scenario concerning a form of speculative execution borrowed from [31]. In this scenario, value speculation is used as a mechanism for increasing parallelism, hence system performance, by predicting values of data dependencies between tasks. Whenever a value prediction is incorrect, corrective actions must be taken in order to re-execute the data consumer code with the correct data value. In this regard, as shown in [14] for a shared-memory setting, reversible execution can permit to relieve programmers from the burden of properly undoing the actions subsequent to an incorrect prediction. Here, we tailor the scenario to the channel-based communication model of session-based programming, and show how
our rollback safety checking supports programmers in identifying erroneous rollback recovery settings.

In the producer-consumer scenario depicted in Fig. 13(a) the session participant \( P \) produces a value and the participant \( C \) consumes it. The data dependence between \( P \) and \( C \) serialises their executions, thus forcing \( C \) to wait for the completion of the value production that requires a fairly long time. In the scenario in Fig. 13(b), instead, \( C \) enacts a speculative behaviour, as it predicts ahead of time the value computed by \( P \) from a partial information. By using the predicted value, \( C \) can execute speculatively and concurrently with \( P \). When \( P \) completes the production, \( C \) validates the prediction by comparing the actual value computed by \( P \) and the predicted one; if the prediction is precise, we gain performance because the execution of \( C \) and \( P \) overlapped in time, otherwise rollback is used to move \( C \) and \( P \) back to a state that precedes the speculative behaviour, in order to re-execute \( C \) using the correct value. The behaviours of \( C \) and \( P \) can be recursively defined in order to repeat the overall execution once a value is correctly consumed.

The scenario informally described above is rendered in cherry-pi as \( \text{start}(x).P_C | \text{start}(y).P_P \), where the consumer and producer processes are:

\[
P_C = \mu X. x!<<(\text{req})>>. x \triangleright \{ l_{\text{spec}} : x?((x_{\text{partial}} : \text{str}).x?((x_{\text{final}} : \text{str}). \text{if } (\text{compare}(x_{\text{partial}}, x_{\text{final}})) \text{ then roll else commit} \ X,) \}
\]

\[
P_P = \mu Y. y?<<(\text{req})>>.
\text{if } (\text{eval}(\text{req})) \text{ then } y \leftarrow l_{\text{spec}}. y!<<(\text{partial}(\text{req})). y!<<(\text{final}(\text{req})). Y.
\text{else } y \leftarrow l_{\text{nonSpec}}. y!<<(\text{compute}(\text{req})). Y.
\]

The producer evaluates each consumer's request in order to establish whether to provide directly the produced value or the partial information for the prediction. In the former case the consumer commits the session and both participants restart, while in the latter one the consumer commits or rolls back depending on the result of the comparison between the predicted value and the produced one.

According to our compliance check, the above collaboration is rollback safe. In case of incorrect prediction the session execution is moved back to the last checkpoint, corresponding to the successfully consumption of the previous requested value. When the producer receives again the same request, it can immediately send the already produced
value. In case the first prediction is wrong, and hence no commit action is performed by the consumer yet, according to the \texttt{cherry-pi} semantics the checkpoint corresponds to the beginning of the session.

Let us consider instead the case of a producer that commits each time a value production is completed, which could apparently seem a reasonable behaviour from the producer side. The resulting collaboration, however, is not rollback safe: while the commit action in the non-speculative case does not affect the compliance between the two session participants, the other commit action overwrites the checkpoint set by the consumer, making it impossible to re-execute the consumer with the correct value. This situation, undesirable for the consumer, is detected by our compliance check.

\section{Extension to multiparty sessions}

We discuss in this section how to extend \texttt{cherry-pi}, its type discipline, the compliance checking and the related results, to multiparty sessions \cite{17}.

The base sets for the multiparty syntax of \texttt{cherry-pi} are the same of the binary case, except for \textit{session endpoints}, which now are denoted by \texttt{s[p]}, with \texttt{p,q} ranging over \textit{roles} (represented as natural numbers). Thus, \textit{session identifiers} \texttt{r} now range over session endpoints \texttt{s[p]} or variables \texttt{x}. The runtime syntax of multiparty \texttt{cherry-pi} is defined by the grammar in Fig.\ref{fig:14} where expressions \texttt{e} are defined as in the binary case (with values that extends to multiparty session endpoints). Primitive \texttt{bar[p][x].P} initiates a new session through identifier \texttt{a} on the other multiple participants, each one of the form \texttt{ar[q][x].P} where \texttt{1 \leq q \leq p - 1}. Variable \texttt{x} will be substituted with the session endpoint used for the interactions inside the established session. Primitive \texttt{rar[p][e].P} denotes the intention of sending a value to role \texttt{p}; similarly, process \texttt{rar[p]?[y:S].P} denotes the intention of receiving a value from role \texttt{p}. Selection and branching are extended in a similar way.

As usual the operational semantics is given in terms of a structural congruence and of a reduction relation. The rules defining the structural congruence are standard, while the forward and backward reduction relations are given by the rules in Fig.\ref{fig:15}. We comment on salient points. Rule [M-F-CON] synchronously initiates a session by requiring all session endpoints be present for a forward reduction, where each role \texttt{p} creates a session endpoint \texttt{s[p]} on a fresh session channel \texttt{s}. The participant with the maximum role is responsible for requesting a session initiation. Rule [M-F-COM] defines how a party with role \texttt{p} synchronously sends a value to the receiving party with role \texttt{q}. Rules [M-F-CMT], [M-B-RLL] and [M-B-ABT] are similar to those of the binary case, and affect all participants within the considered session.

The syntax of session types extends to multiparty as shown in Fig.\ref{fig:16} The session types for output (\texttt{[p][q][e](S).T}) and input (\texttt{[p][q]?[y:S].T}) are extended with information about the interacting roles; selection and branching types are similarly extended. In the
s[p](q)∈(c).P \xrightarrow{a[p](q)} P \quad \text{[M-P-SND]} \quad s[p](q)∈(y:S).P \xrightarrow{a[p](q)} P \quad \text{[M-P-RCV]}

Once a role is unknown, it is used a placeholder \_ to be filled with a given role; \_·p denotes the type obtained from \_ by filling all its placeholders with the role p. The \_·p type system extends accordingly, as shown in Fig. 17.
\[ [p][q]!\langle S \rangle.T \xrightarrow{[p][q]!\langle S \rangle} T \quad \text{[M-TS-SND]} \]
\[ [p][q]?\langle S \rangle.T \xrightarrow{[p][q]?\langle S \rangle} T \quad \text{[M-TS-Rcv]} \]

\[\begin{align*}
T_i \quad &\xrightarrow{\text{[M-TS-Com]}} \quad T_i' \\
(T^I) : \langle \hat{U}_i \rangle \bullet T_i \parallel \prod_{h \in \ell \setminus (i,j)} \langle \hat{U}_h \rangle \bullet T_h \\
&\quad \xrightarrow{\text{[M-TS-Cmt]}} \quad \langle \hat{U}_i' \rangle \bullet T_i' \parallel \prod_{h \in \ell \setminus (i,j)} \langle \hat{U}_h' \rangle \bullet T_h \\
T_i \quad &\xrightarrow{\text{[M-TS-Rll]}} \quad T_i' \\
(T^I) : \langle \hat{U}_i \rangle \bullet T_i \parallel \prod_{h \in \ell \setminus (i)} \langle \hat{U}_h \rangle \bullet T_h \\
&\quad \xrightarrow{\text{[M-TS-Abt]}} \quad (T^I) : \langle \hat{U}_i' \rangle \bullet T_i' \parallel \prod_{h \in \ell} \langle \hat{T}_h \rangle \bullet T_h \\
\end{align*}\]

![Fig. 18. Multiparty cherry-pi type semantics (excerpt of rules, where \( I = \{1, \ldots, n\} \), \( i, j \in I \), and \( T^I \) denotes \( T^1, \ldots, T^n \)).](image)

The semantics of type configurations is defined only for filled types. Semantic rules in Fig. 18 are the natural extension of those for the binary case. Rule [M-TS-Com] shows that communication affects only the two interacting parties, without modifying any checkpoint. Rule [M-TS-Cmt] sets the checkpoint of the committing party and sets an imposed checkpoint for each other party that has performed at least an action from its current checkpoint. Rule [M-TS-Rll] rolls all parties back to their checkpoints, provided that the checkdpoint of the party requesting the rollback is not imposed. Rule [M-TS-Abt] brings all parties back to the initial configuration.

Our notion of rollback safety, and the related compliance relation, extend to multiparty sessions as follows. Notice that types \( T_1 \) in Def. 4 contain placeholders, while in Def. 5 all types are filled.

**Definition 4 (Multiparty Rollback safety).** Let \( C \) be an initial collaboration, then \( C \) is rollback safe (shortened r-safe) if \( C \vdash A \) and for each \( n \)-tuple \( a[n] : T_n, \ldots, a[1] : T_1 \) in \( A \) we have \( \parallel \langle T_n \cdot n, \ldots, T_1 \cdot 1 \rangle \).

**Definition 5 (Compliance for Multiparty Sessions).** Types \( T_1, \ldots, T_n \) are compliant, written \( \parallel \langle T_1, \ldots, T_n \rangle \), if \( \parallel \langle T_1, \ldots, T_n \rangle : \langle T_1 \rangle \bullet T_1, \ldots, \langle T_n \rangle \bullet T_n \rangle \). Relation \( \parallel \) on type configurations is defined as follows: \( \parallel \langle (T^1, \ldots, T^n) : \langle U_1 \rangle \bullet T_1, \ldots, \langle U_n \rangle \bullet T_n \rangle \) holds if for any \( U'_1, T'_1, \ldots, U'_n, T'_n \) such that \( (T^1, \ldots, T^n) : \prod_{h \in \ell \setminus (1, \ldots, n)} \langle U'_h \rangle \bullet T'_h \) we have that \( T'_1 = \ldots = T'_n = \text{end} \).

All notions and concepts of our rollback recovery approach smoothly extend to the multiparty case. As consequence, all properties in Sec. 5 still hold in the extended setting;
We report below just the key theorem concerning session progress.

**Theorem 7 (Multiparty session progress).** Let $C$ be a $r$-safe collaboration of the form $(\bar{a}[n](x).P_n \mid \prod_{i \in \{1, \ldots, n-1\}} a[i](x).P_i)$. If $C \rightarrow^* C'$ then either $C' \rightarrow C''$ for some $C''$ or $C' \equiv (\nu s : C) \prod_{i \in \{1, \ldots, n\}} (\tilde{Q}_i) \rightarrow^* 0$ for some $\tilde{Q}_1, \ldots, \tilde{Q}_n$.

8 Conclusion and related work

This paper proposes rollback recovery primitives for session-based programming, which permit checkpointing session participants, rolling back to the last committed checkpoints, or completely aborting a session. These primitives come together with session typing, enabling a design time compliance check that statically ensures checkpoint persistency properties (Lemma 1 and Theorem 3) and session soundness (Theorems 4 and 5), which differ from the properties of the existing (fully) reversible calculi. Our compliance check has been implemented in the MAUDE framework. We have first illustrated our rollback recovery approach in the binary sessions setting, and then we have shown how it smoothly extends to the multiparty case.

In the literature we can distinguish two ways of dealing with rollback: either using explicit rollbacks and implicit commits [20], or by using explicit commits and spontaneous aborts [11,36]. Differently from these works, we have introduced a way to control reversibility by both triggering it and limiting its scope. Reversibility is triggered by means of an explicit rollback primitive (as in [20]), while explicit commits limit the scope of potential future reverse executions (as in [11,36]). Differently from [11,36], commit does not require any synchronisation, as it is a local decision. This could lead to runtime misbehaviours where a process willing to roll back to its last checkpoint reaches a point which has been imposed by another participant of the session. Our type discipline allows for ruling out programs which may lead to such undesired states.

Reversibility in behavioural types has been studied in different formalisms: behavioural contracts [13], binary session types [27], multi party session types [26,32,33], and global graphs [28,13]. In [13] choices can be seen as implicit checkpoints and the system can go back to a previous choice and try another branch. In [1] rollback is triggered non-deterministically (e.g., it may happen at any time during the execution), while in [3] it is triggered by the system only when the computation is stuck (e.g., the client cannot further continue). Types information is used at runtime by monitors, for binary [27] and multiparty [26] settings, to keep track of the computational history of the system. There, reversibility is uncontrolled, and each computational step can be undone. In [5] global types are enriched with history information, and choices are seen as labelled checkpoints. At any moment, the party who decided which branch to take in a choice may decide to revert it, forcing the entire system to go back to a point prior to the choice. Hence, rollback is confined inside choices and it is spontaneous. Checkpoints are not seen as commits, and a rollback can bring the system to a state prior to several checkpoints. In [32] an uncontrolled reversible variant of session $\pi$-calculus is presented, while [33] studies different notions of reversibility for both binary and multiparty single sessions. In [28,13] global graphs are extended with conditions on branches. These conditions at
runtime can trigger coordinated rollbacks to revert a distributed choice. Reversibility is confined into branches of a distributed choice and not all the computational steps are reversible; inputs, in fact, are irreversible unless they are inside an ongoing loop. Hence, it is not always possible to revert system computations.

We detach from these works in several ways. First our checkpoint facility is explicit and checkpointing is not relegated to choices: the programmer can decide at any point when to commit. This because the programmer may be interested in committing, besides choice points, a series of interactions (e.g., to make a payment irreversible). Thus, once a commit is taken, the system cannot revert to a state prior to it. Our rollback facility is explicit, meaning that it is the programmer who deliberately triggers a rollback, and it is not the system to decide when to go back. Our extension to the multiparty setting is natural and does not rely on a formalism to describe the global view of the system.

Concerning our compliance check, the relation at its basis resembles those given in [123], which however are defined for different rollback recovery approaches based on implicit checkpoints (see comments above). Like these compliance checks, also the check we propose is decidable. Concerning the MAUDE implementation of the compliance check, we have followed the approach of the seminal work by Verdejo and Martí-Oliet [35], providing the state-of-the-art implementation of CCS in MAUDE. Along the same line, many other MAUDE implementations of formalisms and languages have been proposed, such as BPMN [8], Twitlang [23], SCEL [4], and QFLan [34].

As future work, we plan to extend our approach to deal with sessions where parties can interleave interactions performed along different sessions. This requires to deal with subordinate sessions, which may affect enclosing sessions by performing, e.g., commit actions that make some interaction of the enclosing sessions irreversible, similarly to nested transactions [37]. To tackle this issue it would be necessary to extend the notion of compliance relation to take into account possible partial commits (in case of nested sub-sessions) that could be undone at the top level if a rollback is performed.

References

A Proofs

A.1 Decidability result

\textbf{Theorem 1}. Let \( T_1 \) and \( T_2 \) be two session types, checking if \( T_1 \parallel T_2 \) holds is decidable.

\textbf{Proof.} By Def. 1 checking \( T_1 \parallel T_2 \) consists in checking that types \( T'_1 \) and \( T'_2 \) of each configuration \((T_1, T_2) : \langle U'_1 \rangle \bullet T'_1 \parallel \langle U'_2 \rangle \bullet T'_2 \) such that \((T_1, T_2) : \langle T_1 \rangle \bullet T_1 \parallel \langle T_2 \rangle \bullet T_2 \rightarrow^* (T_1, T_2) : \langle U'_1 \rangle \bullet T'_1 \parallel \langle U'_2 \rangle \bullet T'_2 \rightarrow (i.e., type configurations that are reachable from the initial one and that cannot further evolve) are end types. Thus, to prove that the compliance check is decidable we have to show that the number of these reachable configurations is finite. Let us consider the transition system \( TS = \langle S, R \rangle \) associated to the type configuration \( t = (T_1, T_2) : \langle T_1 \rangle \bullet T_1 \parallel \langle T_2 \rangle \bullet T_2 \) by the reduction semantics of types (Fig. 11): the set \( S \) of states corresponds to the set of type configurations reachable from \( t \), i.e. \( S = \{ t' \mid t \rightarrow^* t' \} \), while the set \( R \) of system transitions corresponds to set of the type reductions involving configurations in \( S \), i.e. \( R = \{ (t', t'') \in S \times S \mid t' \rightarrow t'' \} \). Hence, checking \( T_1 \parallel T_2 \) boils down to check the type configurations corresponding to the leaves (i.e., states without outgoing transitions) of \( TS \). Specifically, given a leaf of \( TS \) corresponding to \((T_1, T_2) : \langle V_1 \rangle \bullet V_2 \parallel \langle V_3 \rangle \bullet V_4 \), we have to check if \( V_2 = V_4 = \text{end} \). The decidability of this check therefore depends on the finiteness of \( TS \). This result is ensured by the fact that: (i) backward reductions connect states of \( TS \) only to previously visited states of \( TS \) (Theorem 2), and (ii) our language of types (Fig. 7) corresponds to a CCS-like process algebra without static operators (i.e., parallel and restriction operators) within recursion (see 29 Sec. 7.5).

A.2 Reversibility results

We can instrument our semantics in order to carry the information on which session the reduction is taking place. Hence we will indicate with \( C \rightsquigarrow C' \) the fact that the reduction is taking place on session \( s \). We can show that sessions are independent:

\textbf{Lemma 3 (Swap Lemma)}. Let \( C \) be a collaboration and \( s \) and \( r \) two sessions. If \( C \rightsquigarrow s C_1 \rightsquigarrow^r C_2 \) then there exists a collaboration \( C_3 \) such that \( C \rightsquigarrow^r C_3 \rightsquigarrow s C_2 \).

\textbf{Proof.} By case analysis on the reductions \( \rightsquigarrow s \) and \( \rightsquigarrow^r \).

\textbf{Lemma 4}. Let \( C \) be a collaboration. If \( C \rightsquigarrow^* C_1 \), then for any session \( s \) in \( C_1 \) there exists a collaboration \( C_0 \) such that \( C \rightsquigarrow^* C_0 \rightsquigarrow^* C_1 \) and \( s \) is never used in the trace \( C \rightarrow \) \( C_0 \).

\textbf{Proof.} By induction on the number \( n \) of reduction on \( s \). If there are no reductions then the thesis banally holds. Otherwise we can take the very last reduction on \( s \), that is the closest one to \( C_1 \) and iteratively apply Lemma 3 in order to bring it to the very end. Then we can conclude by induction on a trace with less occurrences of reductions on \( s \).
Thanks to Lemma 4, we can rearrange any trace as a sequence of independent sessions. Moreover given an initial collaboration \( C \) for any reachable collaboration \( C_1 \) and session \( s \) such that \( C \xrightarrow{\ast} C_1 \xrightarrow{\ast} \ast \) and \( s \notin \ast \), we indicate \( C_1 \) as the initial collaboration for \( s \). This will allow us to focus just on a single session, say \( s \), and to consider collaboration initial for \( s \) without losing of generality.

**Lemma 5.** Let \( C \) be an initial collaboration such that \( C \xrightarrow{\ast} C_1 \). If \( C_1 \xrightarrow{\text{abt}} C_2 \) then \( C_2 \equiv C \).

**Proof.** Since \( C \) is initial, without losing of generality we can assume \( C \equiv \bar{a}(x_1).P_1 \mid a(x_2).P_2 \).

The first reduction of \( C \xrightarrow{\ast} C_1 \) has to be an application of rule [F-CON], that is
\[
C \xrightarrow{\nu s : (\bar{a}(x_1).P_1 \mid a(x_2).P_2)} \langle (P_1[s/x_1] \bullet P_1[s/x_1]) \mid (P_2[s/x_2] \bullet P_2[s/x_2]) \rangle = C'
\]
and, by hypothesis, \( C' \xrightarrow{\ast} C_1 \).

Now, no matter the shape of processes in \( C_1 \) by applying rule [B-ABT], and possibly [B-STR], we will go back to \( C \), that is \( C_1 \xrightarrow{\text{abt}} C \), as desired.

The following lemma states that a rollback leads back to the last committed checkpoint.

**Lemma 6.** Let \( C \) be a reachable collaboration, such that \( C \xrightarrow{\text{cmt}} C_1 \). If \( C_1 \xrightarrow{\ast} C_2 \xrightarrow{\text{roll}} C_3 \) and there is no commit in \( C_1 \xrightarrow{\ast} C_2 \), then \( C_3 \equiv C_1 \).

**Proof.** Since \( C \) is a reachable collaboration, this implies it has been generated from an initial collaboration \( C_0 \). Without losing of generality, similarly to the Lemma 5’s proof, we can assume \( C \equiv \nu s : C_0)(P_1 \bullet P_2 \mid Q_1 
arrow Q_2) \mid C^o \). Therefore, we have that \( C_1 \equiv \nu s : C_0)(P_2 \bullet P_2 \mid Q_2 \bullet Q_2) \mid C^o \). By hypothesis, there is no commits in \( C_1 \xrightarrow{\ast} C_2 \), and this implies that the log part of the \( C_1 \) will never change. Hence, we have that \( C_2 \equiv \nu s : C_0)(P_2 \bullet P_2 \mid Q_2 \bullet Q_2) \mid C^o \) for some processes \( P \) and \( Q \). By applying [B-RLL] and [B-PAR] we have that \( C_2 \xrightarrow{\text{roll}} (\nu s : C_0)(P_2 \bullet P_2 \mid Q_2 \bullet Q_2) \mid C^o \equiv C_1 \), as desired.

Any reachable collaboration is also a forward only reachable collaboration. Formally:

**Theorem 2.** Let \( C_0 \) be an initial collaboration. If \( C_0 \xrightarrow{\ast} C_1 \) then \( C_0 \xrightarrow{\ast} C_1 \).

**Proof.** By induction on the number \( n \) of backward reductions contained into \( C_0 \xrightarrow{\ast} C_1 \). The base case \( (n = 0) \) trivially holds. In the inductive case, let us take the backward reduction which is the nearest to \( C_0 \). That is:
\[
C_0 \xrightarrow{\ast} C' \xrightarrow{\ast} C'' \xrightarrow{\ast} C_1
\]
Depending whether it is an \( \text{abt} \) or a \( \text{roll} \) we can apply respectively Lemma 5 or Lemma 6 to obtain a forward trace of the form

\[
C_0 \rightarrow^* C'' \rightarrow^* C_1
\]

and we can conclude by applying the inductive hypothesis on the obtained trace which contains less backward moves with respect to the original one.

\textbf{Lemma 1.} Let \( C_1 \) and \( C_2 \) be two reachable collaborations. \( C_1 \xrightarrow{\text{abt}} C_2 \) then \( C_2 \rightarrow^* C_1 \).

\textbf{Proof.} Since \( C_1 \) is a reachable collaboration, we have that there exists an initial collaboration \( C_0 \) such that \( C_0 \rightarrow^* C_1 \). By applying Theorem 2 we can rearrange the trace such that it contains just forward transitions as follows

\[
C_0 \rightarrow^* C_1
\]

If the backward reduction is obtained by applying \([ \text{B-ABT} ]\), by Lemma 5 we have \( C_2 \equiv C_0 \), from which the thesis trivially follows. Instead, if the backward reduction is obtained by applying \([ \text{B-RLL} ]\), we proceed by case analysis depending on the presence of commit reductions in the trace. If they are present, we select the last of such commit, that is we can decompose the trace in the following way:

\[
C_0 \rightarrow^* \text{cmt} \rightarrow^* C_1 \xrightarrow{\text{roll}} C_2
\]

and by applying Lemma 6 we have that \( C_2 \rightarrow^* \equiv C_1 \) as desired.

In the case there is no commit in the trace, without losing of generality we can assume

\[
C_0 \equiv \tilde{a}(x_1).P_1 | a(x_2).P_2 | C^\circ \\
C_0 \rightarrow (\nu s : C_0)((P_2) \cdot P_2 | (Q_2) \cdot Q_2) | C^\circ
\]

By rule \([ \text{B-RLL} ]\), \( C_2 \equiv (\nu s : C_0)((P_2) \cdot P_2 | (Q_2) \cdot Q_2) | C^\circ \). Thus, we can conclude by noticing that \( C_0 \rightarrow C_2 \) must be the first reduction in \( C_0 \rightarrow^* C_1 \).

\textbf{Lemma 2.} Let \( C \) be a reachable collaboration. If \( C \xrightarrow{\text{roll}} C' \) and \( C \xrightarrow{\text{roll}} C'' \) then \( C' \equiv C'' \).

\textbf{Proof.} Since \( C \) is a reachable collaboration, it is has been generated by an initial collaboration \( C_0 \) of the form \( C_0 = \tilde{a}(x).P_1 | a(x).P_2 \), and by Theorem 2 we have that \( C_0 \rightarrow^* C \). We distinguish two cases, whether in the trace there has been at least one commit or not. In the first case, we can decompose the trace in such a way to single out the last commit as follows:

\[
C_0 \rightarrow^* C_\text{cmt} \rightarrow^* C
\]

so that in the reduction \( C_\text{cmt} \rightarrow C \) there is no commit. If from \( C \) the rollbacks \( C \xrightarrow{\text{roll}} C' \) and \( C \xrightarrow{\text{roll}} C'' \) are triggered by the same process, the thesis trivially follows. In the other case, we have that:

\[
C_\text{cmt} \equiv (\nu s : C_0)((P_2) \cdot P_2 | (Q_2) \cdot Q_2) | C^\circ
\]
with both $P_2$ and $Q_2$ are able to trigger a rollback. If the roll action is executed by $P_2$ we have that
\[(\nu s : C_0)(\langle P_1 \bullet P_2 \mid \langle Q_1 \bullet Q_2 \rangle) \mid C^o \xrightarrow{roll} (\nu s : C_0)(\langle P_1 \bullet P_1 \mid \langle Q_1 \bullet Q_1 \rangle) \mid C^o = C''\]
If the roll is triggered by $Q_2$ we have that
\[(\nu s : C_0)(\langle P_1 \bullet P_2 \mid \langle Q_1 \bullet Q_2 \rangle) \mid C^o \xrightarrow{roll} (\nu s : C_0)(\langle P_1 \bullet P_1 \mid \langle Q_1 \bullet Q_1 \rangle) \mid C^o = C''\]
And we can conclude by noticing that $C' \equiv C''$, as desired.

**Theorem 3** Let $C$ be a reachable collaboration. If $C \xrightarrow{cmt} C'$ then there exists no $C''$ such that $C' \xrightarrow{* roll} C''$ and $C'' \xrightarrow{+} C$.

**Proof.** We proceed by contradiction. Suppose that there exists $C''$ such that $C' \xrightarrow{* roll} C''$ and $C'' \xrightarrow{+} C$. Since $C$ is a reachable collaboration, thanks to Theorem 2 we have that there exists an initial collaboration $C_0$ such that $C_0 \xrightarrow{*} C \xrightarrow{cmt} C'$. Since a rollback brings back the collaboration to a point before a commit, this means it has been restored a commit previous to the last one (by [B-R1-L], indeed, only processes stored in logs can be committed). This implies that there exist at least two different commits in the trace such that
\[C_0 \xrightarrow{*} C_{cmt} \xrightarrow{*} C' \xrightarrow{* roll} C''\]
with $C'' = C_{cmt}$. We have that $C'' \xrightarrow{+} C \xrightarrow{cmt} C' \xrightarrow{*} C_{rl}$, where $C_{rl}$ is the collaboration that perform the roll reduction. Now since the last commit has been done by $C$, supposing that the commit is triggered by $P_c$, which evolves to $P'_c$ in doing that, we have that:
\[C \equiv (\nu s : C_0)(\langle P \bullet P_c \mid \langle Q \bullet Q_c \rangle) \quad \text{and} \quad C' = (\nu s : C_0)(\langle P'_c \bullet P'_c \mid \langle Q_c \bullet Q_c \rangle)\]
Now, by hypothesis we have that $C' \xrightarrow{*} C_{rl}$ without any commit being present in the trace, hence:
\[C_{rl} \equiv (\nu s : C_0)(\langle P'_c \bullet P_{rl} \mid \langle Q_c \bullet Q_{rl} \rangle)\]
By hypothesis, from $C_{rl}$ a rollback is possible. Regardless the rollback is triggered by $P_{rl}$ or $Q_{rl}$, we have that $C_{rl} \xrightarrow{roll} C'$. Now, from $C'$ we cannot reach $C' \xrightarrow{+} C$, as $C'$ is derived from $C$ and the rollback can only bring the collaboration back to $C'$. This violates the hypothesis, and hence we conclude.

### A.3 Soundness results

To prove our soundness results, we need to introduce some auxiliary lemmas, which rely on the following definitions:
- a process $\tilde{P}$ and a type $\tilde{T}$ are in **checkpoint accordance** if $\tilde{P} = P$ implies $\tilde{T} = T$ and $\tilde{P} = P$ implies $\tilde{T} = T$;
- let $\ell$ a process label, its **dual label** $\tilde{\ell}$ is defined as follows: $k\ell(x) = k'$, $k(\ell(x)) = k$, $k < l = k \succ l$, $k \succ l = k \prec l$; this notion of duality straightforwardly extends to type labels;
Proof. We have two cases depending whether the reduction → has forward or backward direction.

\((\Leftarrow = \rightarrow)\). From rule [F-Res], we have \(\langle \bar{P}_1 \rangle \cdot P_2 \mid \langle \bar{Q}_1 \rangle \cdot Q_2 \rightarrow \langle \bar{P}_1' \rangle \cdot P_2' \mid \langle \bar{Q}_1' \rangle \cdot Q_2'\). We prove the result by case analysis on the last rule applied in the inference of the above reduction.

- [F-Com]. In this case we have \(P_2 \xrightarrow{x!v} P_2', Q_2 \xrightarrow{x!v} Q_2', \) with \(Q_2' = Q_2'[u/v]\). Thus, \(P_2' = \bar{P}_1' \) and \(Q_2' = \bar{Q}_1'\). By rule [T-SND], we have that \(T_2 = [S].T_2', \) with \(T_2' \leftarrow e \cdot S\), (hence \(T_2' \leftarrow e \cdot S\)) and \(T_2' \leftarrow P_2'[u/v]\). Similarly, by rule [T-RCV], we have that \(U_2 = [S]'\). Thus, \(T_2' \leftarrow Q_2'[u/v]\) and \(Q_2' \leftarrow Q_2'[u/v]\) (hence \(T_2' \leftarrow Q_2'[u/v]\)). By rules [T-NCVD] and [T-RCV], we get \(\langle \bar{T}_1 \rangle' \cdot T_2' \parallel \langle \bar{U}_1 \rangle' \cdot U_2' \rightarrow \langle \bar{T}_1' \rangle \cdot T_2' \parallel \langle \bar{U}_1 \rangle' \cdot U_2'\). Now, reasoning by contradiction, let us suppose that \(S \neq S'\). Thus, the term \((T, T') : \langle \bar{T}_1 \rangle \cdot T_2' \parallel \langle \bar{U}_1 \rangle' \cdot U_2' \rightarrow \langle \bar{T}_1' \rangle \cdot T_2' \parallel \langle \bar{U}_1 \rangle' \cdot U_2'\). By Def. [Rollback Recovery] \(T \rightarrow T'\) implies \(T_2 = U_2 = end\), which is a contradiction. Therefore, it holds that \(S = S'\). Hence, by applying rule [TS-NCVD] we can conclude.

- [F-LAB], [E-CMT1] and [E-CMT2]. Similar to the previous case.

- [F-STR]. The use of \(\equiv\) leads us back to one of the other cases.

- [F-PAR]. In this case we have that \(\langle \bar{P}_1 \rangle \cdot P_2 \rightarrow \langle \bar{P}_1' \rangle \cdot P_2'\). Since this transition involves only one log term, it can be inferred only by applying rule [P-IF], from which we have \(P_2' \leftarrow P_2' \) and \(P_1' = P_1\). By rule [P-IR-TF] (the case of rule [P-IF] is similar), we have \(P_2'[x/s] = \text{if } e \text{ then } P_2'[x/s] \text{ else } R\) with \(e \rightarrow \langle P_1' \rangle\). By rule [T-IF] we get \(T_2' \Rightarrow V\) and \(\bar{Q}_2' \leftarrow P_2'[x/s]\). By rule [TS-IF], we have \(T_2' \Rightarrow T_2'\). By applying rule [TS-NCVD] we can conclude.

\((\Leftarrow = \equiv)\). From rule [B-Res], we have \(\langle \bar{P}_1 \rangle \cdot P_2 \mid \langle \bar{Q}_1 \rangle \cdot Q_2 \equiv \langle \bar{P}_1' \rangle \cdot P_2' \mid \langle \bar{Q}_1' \rangle \cdot Q_2'\). We prove the result by case analysis on the last rule applied in the inference of the above reduction.
Lemma 8. Let $C = (\tilde{a}(x).P \mid a(y).Q)$ such that $C \Rightarrow \{\tilde{a} : T_1, a : T_2\}$ and $T_1 \Downarrow T_2$. If $C \Rightarrow^* (\nu s : C)(\langle P_1 \rangle \triangleright P_2 \mid \langle Q_1 \rangle \triangleright Q_2)$ and $P_2 \triangleright \Gamma T_2$, then there exist $U_1, U_2, U'_1$, $U'_2, U''_1$ such that $(T_1, T_2) : \langle T_1 \rangle \bullet T_1 \Downarrow T_2 \Rightarrow^* (T_1', T_2) : \langle U_1 \rangle \bullet U'_1 \Downarrow \langle U_2 \rangle \bullet U'_2$ and $U'_1 \triangleright \Gamma T_1$, $U''_1 \triangleright \Gamma T_2$.

Proof. From $C \Rightarrow \{\tilde{a} : T_1, a : T_2\}$, by applying [T-PAR], [T-ACC] and [T-REQ], we have that $\emptyset; \emptyset \vdash T \bullet \bullet x : T_1$ and $\emptyset; \emptyset \vdash Q \bullet \bullet x : T_2$. By applying rule [F-CON] to the collaboration $C$, we obtain $C \Rightarrow (\nu s : C)(\langle P[s/x] \rangle \triangleright P[s/x] \mid \langle Q[s/y] \rangle \triangleright Q[s/y]) = C'$. Now, by repeatedly applying Lemma 7 from $C' \Rightarrow^* (\nu s : C)(\langle P \rangle \triangleright P_2 \mid \langle Q \rangle \triangleright Q_2)$ we get $(T_1, T_2) : \langle T_1 \rangle \bullet T_1 \Downarrow T_2 \Rightarrow^* (T_1', T_2) : \langle U_1 \rangle \bullet U'_1 \Downarrow \langle U_2 \rangle \bullet U'_2$ for some $U_1, U_2, U'_1, U'_2$, with $\Theta_2; T_2 \vdash P_2[x/s] \bullet x : U'_2$. Now, let us consider the transition $P_2 \triangleright \Gamma T_2$. This can be derived only by the application of rules [P-RLL]. Thus, $P_2 = \text{roll}$, from which we have $P_2[x/s] = \text{roll}$. From $\Theta_2; T_2 \vdash \text{roll} \bullet \bullet x : U'_2$, by rule [T-RLL], we get $U'_1 = \text{roll}$. Therefore, by rule [T-STR], we can conclude $U'_1 \triangleright \Gamma T_1$ with $U''_1 = \text{end}$.

The following lemma relates collaboration reductions to type reductions when a \text{com-error} is produced.

Lemma 9. Let $C = (\tilde{a}(x).P \mid a(y).Q)$ such that $C \Rightarrow \{\tilde{a} : T_1, a : T_2\}$ and $T_1 \Downarrow T_2$. If $C \Rightarrow^* (\nu s : C)(\langle P_1 \rangle \triangleright P_2 \mid \langle Q_1 \rangle \triangleright Q_2)$, $P_2 \triangleright \Gamma \ell$ with $\ell$ of the form $\kappa!(v)$, $\kappa?z(x)$, $k = l$ or $k \succ l$, then there exist $U_1, U_2, U'_1, U'_2$ such that $(T_1, T_2) : \langle T_1 \rangle \bullet T_1 \Downarrow T_2 \Rightarrow^* (T_1', T_2) : \langle U_1 \rangle \bullet U'_1 \Downarrow \langle U_2 \rangle \bullet U'_2$ with $P_1$ (resp. $Q_1$) in checkpoint accordance with $U_1$ (resp. $U'_2$), and for all $U''_1$ such that $U_2 \Rightarrow^* U''_1$ we have $U''_1 \triangleright \Gamma T_1$.

Proof. From $C \Rightarrow \{\tilde{a} : T_1, a : T_2\}$, by applying [T-PAR], [T-ACC] and [T-REQ], we have that $\emptyset; \emptyset \vdash P \bullet \bullet x : T_1$ and $\emptyset; \emptyset \vdash Q \bullet \bullet x : T_2$. By applying rule [F-CON] to the collaboration $C$, we obtain $C \Rightarrow (\nu s : C)(\langle P[s/x] \rangle \triangleright P[s/x] \mid \langle Q[s/y] \rangle \triangleright Q[s/y]) = C'$. Now, by repeatedly applying Lemma 7 from $C' \Rightarrow^* (\nu s : C)(\langle P \rangle \triangleright P_2 \mid \langle Q \rangle \triangleright Q_2)$ we get $(T_1, T_2) : \langle T_1 \rangle \bullet T_1 \Downarrow T_2 \Rightarrow^* (T_1', T_2) : \langle U_1 \rangle \bullet U'_1 \Downarrow \langle U_2 \rangle \bullet U'_2$ for some $U_1, U_2, U'_1, U'_2$, with $\Theta_2; T_2 \vdash P_2[x/s] \bullet x : U'_2$ and $\Theta'_2; T'_2 \vdash Q_2[y/s] \bullet y : U'_2$. Now, let us reason by case analysis on the rule for deriving the transition $P_2 \triangleright \Gamma T_2$.

Rule [P-SND]. Thus, $P_2 = \mathcal{S}!(y), P'_2$ and $\ell = \mathcal{S}!(v)$ with $e \downarrow v$. From $\Theta_2; T_2 \vdash P_2[x/s] \bullet x : U'_2$, by rule [T-SND], we get $U'_2 \Rightarrow [S].U''_1$ with $\Gamma \vdash e \bullet S$.

Therefore, by rule [TS-SND], we get $U'_1 \triangleright [S].U''_1$. 
Rule [P-Rcv]. Thus, $P_2 = \bar{s}(y : S). P''_2$ and $\ell = \bar{s}(y)$. From $\Theta_2 : F_2 \vdash P_2[x/s] \rightarrow x : U'_1$, by rule [T-Rcv], we get $U'_1 = ?[S]. U''$. Therefore, by rule [TS-Rcv], we get $U'_1 \rightarrow^{?}[S], U''$. 

Rules [P-Sele] and [P-Brn]. Similar to the previous cases. 

Rule [P-Str]. According to the definition of $\equiv$, the only relevant case is when $P_2 = \mu X.R$. Thus, by rule [P-Str] we have $P_2 \equiv R[\mu X.R/X] = P''_2$, and $P''_2 \not\rightarrow P''_2$ with $P''_2 \equiv P''_2$. If $P''_2$ has not the form $\mu Y.R'$, we can proceed as in one of the cases above, otherwise we repeat this reasoning.

Finally, from $Q_2 \not\rightarrow$, following a similarly reasoning, we can conclude $U'_2 \not\rightarrow^{r}$.

We can now prove our soundness results.

**Theorem 4** If $C$ is a rollback safe collaboration, then $C \not\rightarrow^{*} C[\text{roll\_error}]$. 

**Proof.** The proof proceeds by contradiction. Suppose that there exists an initial collaboration $C$ that is rollback safe and such that $C \not\rightarrow^{*} C[\text{roll\_error}]$. The erroneous collaboration $\text{roll\_error}$ can be only produced by applying rule [E-Rll-2]. Thus, to infer at least one reduction of the sequence $C \not\rightarrow^{*} C[\text{roll\_error}]$, rule [E-Rll-2] must be used. From this, we have that there exists a runtime collaboration $C'' \equiv C'[C''']$, with $C''' = \langle Q_1 \rangle \rightarrow P_1 \mid \langle Q_2 \rangle \rightarrow P_2$, such that $C \not\rightarrow^{*} C'$, $P_1 \rightarrow P''_1$, and $C' \rightarrow C'[\text{roll\_error}] \not\rightarrow^{*} C[\text{roll\_error}]$. By rules [F-Con], [F-Res] and [B-Res], and the fact that the scope of operator (vs : _ ) is statically defined (i.e., neither the operational rules nor $\equiv$ allow scope extension), the term $C''$ can only be the argument of the operator (vs : C'1), i.e. $C'' = C_2 \mid (vs : C_1)$, with $C_1 = a(x).P \mid a(y).Q$ for some $a, x, y, P$ and $Q$. In its own turn, the term (vs : C_1) C'' can only be generated by applying rule [F-Con] from C_1, which must be a subterm of C, i.e. C' $\equiv$ C1 | C'2 for some C'2. Since the scope of (vs : _ ) operator cannot be extended, all reductions performed by terms in parallel with it by applying rules [F-Par] and [B-Par] do not affect the argument of such operator. Therefore, focussing on the subterm of C, by exploiting rules [F-Par] and [B-Par] we can set apart the reductions in C' $\not\rightarrow^{*} C[\text{roll\_error}]$ involving C_1 and its derivatives, thus obtaining C_1 $\not\rightarrow^{*}$ (vs : C_1)C'' $\not\rightarrow$ (vs : C_1)roll\_error.

Now, since C is rollback safe, by Def. [8] we have that C $\not\rightarrow$ A and for all pairs $\delta : V_1$ and $\beta : V_2$ in A we have $V_1 \not\rightarrow V_2$. Since C $\equiv$ C1 | C'2, by rule [T-Par] we obtain C_1 $\not\rightarrow$ A_1 with A_1 $\subseteq$ A. By rules [T-Req] and [T-Acc], we have A_1 = \{a : T_1, a : T_2\}. Since A_1 is a subset of A_2, that is A_1 $\subseteq$ A_2, we have that T_1 $\not\rightarrow$ T_2.

By Lemma [8] we have that there exist U_1, U_2, U'_1, U'_2, U'' such that (T_1, T_2) $\not\rightarrow^{*}$ T_1 || T_2 $\not\rightarrow$ T_1, T_2 $\not\rightarrow^{*}$ (U_1) $\not\rightarrow$ U'_1 || (U'_2) $\not\rightarrow$ U'' $\not\rightarrow$ t and U'_1 $\not\rightarrow$ U'' in Fig. [11]. No rule in Fig. [11] allows the term t to evolve, i.e. t $\not\rightarrow$, because U'_1 can only perform roll and rule [TS-Rll-1] cannot be applied due to the imposed checkpoint U_1. Since T_1 || T_2, by Def. [8] it must hold that U'_1 = end. However, since U'_1 is able to perform an action (as it holds that U'_1 $\not\rightarrow$ end, we get that it cannot be an end type, i.e. U'_1 $\neq$ end, which is a contradiction.
Theorem 5. If \( C \) is a rollback safe collaboration, then \( C \rightarrow^* C[\text{com\_error}] \).

Proof. The proof proceeds by contradiction. Suppose that there exists an initial collaboration \( C \) that is rollback safe and such that \( C \rightarrow^* C[\text{com\_error}] \). The erroneous collaboration \( C[\text{com\_error}] \) can be produced by applying one of the rules [E-COM1], [E-COM2], [E-LAB1] and [E-LAB2]. Let us consider the case [E-COM1], the other cases are similar. Proceeding as in the proof of Theorem 4 without loss of generality we can focus on the subterm \( C_1 = (\bar{a}(x), P \mid a(y), Q) \) of \( C \), such that \( C_1 \rightarrow^* (\nu s : C_1) C'' \rightarrow (\nu s : C_1) \text{com\_error} \), with \( C'' = \left( \langle Q_1 \rangle \bullet P_1 \parallel \langle Q_2 \rangle \bullet P_2 \right) \), and \( C_1 \bullet \{ \bar{a} : T_1, a : T_2 \} \), with \( T_1 \parallel T_2 \).

By Lemma 9 we have that there exist \( U_1, U_2, U'_1, U'_2, U_2'' \) such that \( U_1, U_2, U'_1, U'_2, U_2'' \) \( (T_1, T_2) : \langle T_1 \rangle \bullet T_1 \parallel \langle T_2 \rangle \bullet T_2 \rightarrow^* \langle \bar{U}_1 \rangle \bullet U'_1 \parallel \langle \bar{U}_2 \rangle \bullet U'_2 = t \) with \( \bar{Q}_1 \) (resp. \( \bar{Q}_2 \)) in checkpoint accordance with \( U_1 \) (resp. \( U_2 \)), \( U'_1 \parallel \bar{U}_2 \), and for all \( U''_2 \) such that \( U'_2 \rightarrow^* U''_2 \) we have \( U''_2 \rightarrow [S] U''_1 \). Thus, for all \( U''_2 \) as above, we have \( t \rightarrow^* (T_1, T_2) : \langle \bar{U}_1 \rangle \bullet U'_1 \parallel \langle \bar{U}_2 \rangle \bullet U'_2 = t' \). No rule in Fig. 11 allows the term \( t' \) to evolve, i.e. \( t' \rightarrow \), because \( U_1 \) can only perform \([S] \) and rule [TS-COM] cannot be applied since \( U''_2 \rightarrow [S] U''_1 \). Since \( T_1 \parallel T_2 \), by Def. 1 it must hold that \( U'_1 = \text{end} \). However, since \( U'_1 \) is able to perform an action (as it holds that \( U'_1 \parallel [S] U''_1 \)), we get that it cannot be an end type, i.e. \( U'_1 \neq \text{end} \), which is a contradiction.

We conclude with the session progress theorem.

Theorem 6. Let \( C = (\bar{a}(x_1), P_1 \mid a(x_2), P_2) \) be a rollback safe collaboration. If \( C \rightarrow^* C' \), then either \( C' \rightarrow C'' \) for some \( C'' \) or \( C' \equiv (\nu s : C)(\langle Q_1 \rangle \bullet 0 \parallel \langle Q_2 \rangle \bullet 0) \) for some \( Q_1 \) and \( Q_2 \).

Proof. The proof proceeds by contradiction. Suppose that \( C \) is rollback safe and \( C \rightarrow^* C' \) with \( C' \rightarrow \) and \( C' \equiv (\nu s : C)(\langle Q_1 \rangle \bullet 0 \parallel \langle Q_2 \rangle \bullet 0) \) for any \( Q_1 \) and \( Q_2 \). The only situations that prevents \( C'' \) from progressing are \( C'' = C[\text{roll\_error}] \) and \( C' = C[\text{com\_error}] \). However, from Theorems 4 and 5 respectively, we have \( C' \neq C[\text{roll\_error}] \) and \( C' \neq C[\text{com\_error}] \), which is a contradiction.

A.4 Multiparty session results

All notions and concepts of our rollback recovery approach smoothly extend to the multiparty case. As consequence, all properties in Sec. 5 still hold in the extended setting. Their proofs indeed follow the same structure of the binary case and only differ for the technicalities concerning the extended definitions.

Decidability result Theorem 1. Let \( T_1, \ldots, T_n \) be filled multiparty session types, checking if \( \parallel (T_1, \ldots, T_n) \) holds is decidable.

Proof. By definition of Compliance in Sec. 7, checking \( \parallel (T_1, \ldots, T_n) \) consists in checking that types \( T'_1, \ldots, T'_n \) of each configuration \( (T^1, \ldots, T^n) : \prod_{h \in \{1, \ldots, n\}} \langle T'_h \rangle \bullet \)}
Then we can conclude by induction on a trace with less occurrences of reductions on $T_h$ (i.e., type configurations that are reachable from the initial one and that cannot further evolve) are end types. Thus, to prove that the compliance check is decidable we have to show that the number of these reachable configurations is finite. Let us consider the transition system $TS = \langle \mathcal{S}, \mathcal{R} \rangle$ associated to the type configuration $t = (T^1, \ldots, T^n) : \prod_{i \in \{1, \ldots, n\}} \langle T^i \rangle \rightarrow T_h$ by the reduction semantics of types (Fig. 18), the set $\mathcal{S}$ of states corresponds to the set of type configurations reachable from $t$, i.e. $\mathcal{S} = \{ t' \mid t \rightarrow^* t' \}$, while the set $\mathcal{R}$ of system transitions corresponds to set of the type reductions involving configurations in $\mathcal{S}$, i.e. $\mathcal{R} = \{ (t', t'') \in \mathcal{S} \times \mathcal{S} \mid t' \rightarrow t'' \}$. Hence, checking $\equiv (T_1, \ldots, T_n)$ boils down to check the type configurations corresponding to the leaves (i.e., states without outgoing transitions) of $TS$. Specifically, given a leaf of $TS$ corresponding to $t = (T^1, \ldots, T^n) : \prod_{i \in \{1, \ldots, n\}} \langle T^i \rangle \rightarrow V_h$, we have to check if $V_1 = \ldots = V_n = \text{end}$. The decidability of this check therefore depends on the finiteness of $TS$. This result is ensured by the fact that: (i) backward reductions connect states of $TS$ only to previously visited states of $TS$ (Theorem 8), and (ii) our language of types (Fig. ??) corresponds to a CCS-like process algebra without static operators (i.e., parallel and restriction operators) within recursion (see [29, Sec. 7.5]).

**Reversibility results** As in Appendix A.2, we will indicate with $C \xRightarrow{s} C'$ the fact that the reduction is taking place on session $s$.

**Lemma 10 (Swap Lemma).** Let $C$ be a collaboration and $s$ and $r$ two sessions. If $C \xRightarrow{s} C_1 \xrightarrow{r} C_2$ then there exists a collaboration $C_3$ such that $C \xrightarrow{r} C_3 \xRightarrow{s} C_2$.

*Proof.* By case analysis on the reductions $\xRightarrow{s}$ and $\xrightarrow{r}$.

**Lemma 11.** Let $C$ be a collaboration. If $C \rightarrow^* C_1$, then for any session $s$ in $C_1$ there exists a collaboration $C_0$ such that $C \rightarrow^* C_0 \xRightarrow{s} C_1$ and $s$ is never used in the trace $C \rightarrow^* C_0$.

*Proof.* By induction on the number $n$ of reduction on $s$. If there are no reductions then the thesis banally holds. Otherwise we can take the very last reduction on $s$, that is the closest one to $C_1$ and iteratively apply Lemma 10 in order to bring it to the very end. Then we can conclude by induction on a trace with less occurrences of reductions on $s$.

As in Appendix A.2 thanks to Lemma 11 without losing of generality we can focus just on a single session, say $s$, and to consider collaboration initial for $s$.

**Lemma 12.** Let $C$ be an initial collaboration such that $C \rightarrow^* C_1$. If $C_1 \xrightarrow{abt} C_2$ then $C_2 \equiv C$.

*Proof.* Since $C$ is initial, without losing of generality we can assume

$$C \equiv \vec{a}[n](x).P_n \mid \prod_{i \in I} a[i](x).P_i$$

with $I = \{1, \ldots, n - 1\}$. 

Reversion Recovery in Session-based Programming 35
The first reduction of $C \mapsto^* C_1$ has to be an application of rule [M-F-C\ ON], that is

$$C \mapsto (\nu s: (a[n](x).P_n \mid \prod_{i \in I} a[i](x).P_i))$$

$$\langle P_n[s[n]/x] \bullet P_n[s[n]/x] \rangle \prod_{i \in I} \langle P_1[s[i]/x] \bullet P_2[s[i]/x] \rangle = C'$$

and, by hypothesis, $C' \mapsto^* C_1$.

Now, no matter the shape of processes in $C_1$ by applying rule [M-B-A\ BT] and possibly [M-B-STR], we will go back to $C$, that is $C_1 \xrightarrow{\mathsf{abt}} C$, as desired.

**Lemma 13.** Let $C$ be a reachable collaboration, such that $C \xrightarrow{\text{cmt}} C_1$. If $C_1 \xrightarrow{\mathsf{abt}} C_2 \xrightarrow{\mathsf{roll}} C_3 \text{ and there is no commit in } C_1 \mapsto^* C_2$, then $C_3 \equiv C_1$.

**Proof.** As in Lemma 6’s proof, by hypothesis, there is no commits in $C_1 \mapsto^* C_2$, and this implies that the log part of the $C_1$ will never change. Hence, by applying [M-B-R\ LL] and [M-B-P\ AR] to $C_2$ we can conclude.

**Theorem 8.** Let $C_0$ be an initial collaboration. If $C_0 \mapsto^* C_1$ then $C_0 \mapsto^* C_1$.

**Proof.** By induction on the number $n$ of backward reductions contained into $C_0 \mapsto^* C_1$. The base case ($n = 0$) trivially holds. In the inductive case, let us take the backward reduction which is the nearest to $C_0$. That is:

$$C_0 \mapsto^* C' \xrightarrow{\mathsf{abt}} C'' \mapsto^* C_1$$

Depending whether it is an $\xrightarrow{\mathsf{abt}}$ or a $\xrightarrow{\mathsf{roll}}$ we can apply respectively Lemma 12 or Lemma 13 to obtain a forward trace of the form

$$C_0 \mapsto^* C'' \mapsto C_1$$

and we can conclude by applying the inductive hypothesis on the obtained trace which contains less backward moves with respect to the original one.

**Lemma 14.** Let $C_1$ and $C_2$ be two reachable collaborations. $C_1 \xrightarrow{\mathsf{roll}} C_2$ then $C_2 \mapsto^* C_1$.

**Proof.** (Sketch) Since $C_1$ is a reachable collaboration, we have that there exists an initial collaboration $C_0$ such that $C_0 \mapsto^* C_1$. By applying Theorem 8 we can rearrange the trace such that it contains just forward transitions as follows

$$C_0 \mapsto^* C_1$$

If the backward reduction is obtained by applying [M-B-Abt], by Lemma 12 we have $C_2 \equiv C_0$, from which the thesis trivially follows. Instead, if the backward reduction is obtained by applying [M-B-R\ LL], we proceed by case analysis depending on the presence of commit reductions in the trace. If they are present, we select the last of such commit, that is we can decompose the trace in the following way:

$$C_0 \mapsto^* \xrightarrow{\text{cmt}} C_{\text{cmt}} \mapsto^* C_1 \xrightarrow{\mathsf{roll}} C_2$$
and by applying Lemma [13] we have that $C_2 \rightarrow^* C_1$ as desired. In the case there is no commit in the trace, we can conclude by noticing that $C_0 \rightarrow C_2$ must be the first reduction in $C_0 \rightarrow^* C_1$.

**Lemma 15.** Let $C$ be a reachable collaboration. If $C \xrightarrow{\text{roll}} C'$ and $C \xrightarrow{\text{roll}} C''$ then $C' \equiv C''$.

**Proof.** (Sketch) Since $C$ is a reachable collaboration, it has been generated by an initial collaboration $C_0$, and by Theorem [8] we have that $C_0 \rightarrow^* C$. We distinguish two cases, whether in the trace there has been at least one commit or not. In the first case, we can decompose the trace in such a way to single out the last commit as follows:

$$C_0 \rightarrow^* C_{\text{cmt}} \rightarrow^* C$$

so that in the reduction $C_{\text{cmt}} \rightarrow^* C$ there is no commit. If from $C$ the rollbacks $C \xrightarrow{\text{roll}} C'$ and $C \xrightarrow{\text{roll}} C''$ are triggered by the same process, the thesis trivially follows. In the other case, we have that at least two processes, say $P$ and $Q$, are able to trigger a rollback. If the roll action is executed by $P$ we have that $C_{\text{cmt}} \rightarrow^* C_1$. If the roll is triggered by $Q$ we have that $C_{\text{cmt}} \rightarrow^* C_2$. And we can conclude by noticing that $C_1 \equiv C_2$, as desired.

**Theorem 3.** Let $C$ be a reachable collaboration. If $C \xrightarrow{\text{cmt}} C'$ then there exists no $C''$ such that $C' \rightarrow^* \xrightarrow{\text{roll}} C''$ and $C'' \rightarrow^* C$.

**Proof.** We proceed by contradiction. Suppose that there exists $C''$ such that $C' \rightarrow^* \xrightarrow{\text{roll}} C''$ and $C'' \rightarrow^* C$. Since $C$ is a reachable collaboration, thanks to Theorem [8] we have that there exists an initial collaboration $C_0$ such that $C_0 \rightarrow^* C \xrightarrow{\text{cmt}} C'$. Since a rollback brings back the collaboration to a point before a commit, this means it has been restored a commit previous to the last one (by [M-B-RLL], indeed, only processes stored in logs can be committed). This implies that there exist at least two different commits in the trace such that

$$C_0 \rightarrow^* C_{\text{cmt}} \rightarrow^* C \xrightarrow{\text{cmt}} C' \rightarrow^* \xrightarrow{\text{roll}} C''$$

with $C'' = C_{\text{cmt}}$. We have that $C'' \rightarrow^* C \xrightarrow{\text{cmt}} C' \rightarrow^* C_{\text{rl}}$, where $C_{\text{rl}}$ is the collaboration that perform the roll reduction. Now since the last commit has been done by $C$, supposing that the commit is triggered by $P^c$, which evolves to $P^{c'}$ in doing that, we have that:

$$C \equiv (\nu s : C_0)(\langle P \rangle \bullet P^c) \prod_{i \in I} (Q_i^c \bullet Q_i^c)$$

and

$$C' = (\nu s : C_0)(\langle P^{c'} \rangle \bullet P^{c'}) \prod_{i \in I} (Q_i^{c'} \bullet Q_i^{c'})$$

Now, by hypothesis we have that $C' \rightarrow^* C_{\text{rl}}$ without any commit being present in the trace, hence:

$$C_{\text{rl}} \equiv (\nu s : C_0)(\langle P^{c'} \rangle \bullet P^{rl}) \prod_{i \in I} (Q_i^{c'} \bullet Q_i^{rl})$$
By hypothesis, from $C_t$, a rollback is possible. Regardless the rollback is triggered by $P_t$ or $Q_t$, we have that $C_t \xrightarrow{rol} C'$. Now, from $C'$ we cannot reach $C \setminus C'$, as $C'$ is derived from $C$ and the rollback can only bring the collaboration back to $C'$. This violates the hypothesis, and hence we conclude.

**Soundness results**

**Lemma 16.** Let $C = (\nu s : C') \prod_{i \in I} \langle Q_i \rangle \mapsto P_i$ be a reachable collaboration, with $I = \{1, \ldots, n\}$, $C' = (\hat{a}[n], \ldots, a[1]) \mapsto P_i \prod_{i \in I \setminus \{n\}} a[i](x).R_i$, and for all $i \in I (\Theta_i; F_i \vdash Q_i[x/s[i]] \mapsto x : V_i)$, $(\Theta_i; F_i \vdash P_i[x/s[i]] \mapsto x : U_i)$. If $C \mapsto (\nu s : C') \prod_{i \in I} \langle Q'_i \rangle \mapsto P'_i$ then there exist $V'_i$ and $U'_i$ such that $(\Gamma^1, \ldots, \Gamma^n) : \prod_{i \in I} \langle V'_i \rangle \mapsto (T^1, \ldots, T^n) : \prod_{i \in I} \langle U'_i \rangle \mapsto V'_i$ with $Q'_i$ in checkpoint accordance with $V'_i$, and for all $i \in I (\hat{\Theta}_i; F'_i \vdash Q'_i[x/s[i]] \mapsto x : V'_i)$, $(\hat{\Theta}_i; F'_i \vdash P'_i[x/s[i]] \mapsto x : U'_i)$.

**Proof.** (Sketch) We have two cases depending whether the reduction $\mapsto$ has forward or backward direction.

$(\mapsto = \rightarrow)$. From rule [M-F-RES], we have $\prod_{i \in I} \langle Q_i \rangle \mapsto P_i \rightarrow \prod_{i \in I} \langle Q'_i \rangle \mapsto P'_i$. We prove the result by case analysis on the last rule applied in the inference of the above reduction.

$(\mapsto = \leftarrow \leftarrow)$. From rule [B-RES], we have $\prod_{i \in I} \langle Q_i \rangle \mapsto P_i \leftarrow \prod_{i \in I} \langle Q'_i \rangle \mapsto P'_i$. We prove the result by case analysis on the last rule applied in the inference of the above reduction.

**Lemma 17.** Let $C = \hat{a}[n](x).P_n \mapsto \prod_{i \in I \setminus \{n\}} a[i](x).P_i$, with $I = \{1, \ldots, n\}$, such that $C \mapsto \hat{a}[n] : V_n, \ldots, a[1 : V_1]$, $T_i = V_i \cdot \bot$ for all $i \in I$, and $\Rightarrow (T_n, \ldots, T_1)$. If $C \mapsto^* (\nu s : C)(\langle \hat{R}_1 \rangle \mapsto Q_1 \mapsto \prod_{i \in I \setminus \{n\}} \langle \hat{R}_i \rangle \mapsto Q_i)$ and $Q_1 \mapsto Q_i$.

**Proof.** (Sketch) As in Lemma 8’s proof, this result can be obtained by repeatedly applying Lemma 16.

**Lemma 18.** Let $C = \hat{a}[n](x).P_n \mapsto \prod_{i \in I \setminus \{n\}} a[i](x).P_i$, with $I = \{1, \ldots, n\}$, such that $C \mapsto^* (\nu s : C) \prod_{i \in I \setminus \{n\}} \langle \hat{Q}_i \rangle \mapsto P_i$, $P_n \mapsto P'_n$ and there is no $P_j$ with $j \in \{2, \ldots, n\}$ such that $P_j \not\Rightarrow \langle \nu s \rangle s[p][q] \mapsto l$, then there exist $(U_1 : \hat{U}_1) \mapsto (T_1, \ldots, T_n) : \prod_{i \in I} \langle T_i \rangle \mapsto (\Gamma^1, \ldots, \Gamma^n) \langle \hat{U}_i \rangle \langle U'_i \rangle \mapsto \langle U''_i \rangle$ with $\Gamma$ sorting for typing $P'_n$, and for all $U''_i$ such that $U''_i \mapsto^* U''_i$, we have $U''_i \mapsto^* U''_i$.

**Proof.** (Sketch) As in Lemma 9’s proof, this result can be obtained by repeatedly applying Lemma 16 and reasoning by case analysis on the rule for deriving the transition $P_n \mapsto P'_n$. 
**Theorem 9.** If $C$ is a rollback safe collaboration, then $C \rightarrow^* C[\text{roll\_error}]$.

*Proof.* (Sketch) The proof proceeds by contradiction and relies on Lemma[17]

**Theorem 10.** If $C$ is a rollback safe collaboration, then $C \rightarrow^* C[\text{com\_error}]$.

*Proof.* (Sketch) The proof proceeds by contradiction and relies on Lemma[18]

**Theorem 7** Let $C = (a[n](x).P_n \mid \prod_{i \in \{1, \ldots, n-1\}} a[i](x).P_i)$ be a r-safe collaboration. If $C \rightarrow^* C'$ then either $C' \rightarrow C''$ for some $C''$ or $C' \equiv (\nu s : C) \prod_{i \in \{1, \ldots, n\}} Q_i \rightarrow^* 0$ for some $Q_1, \ldots, Q_n$.

*Proof.* The proof proceeds by contradiction. Suppose that $C$ is rollback safe and $C \rightarrow^* C'$ with $C' \rightarrow$ and $C' \equiv (\nu s : C) \prod_{i \in \{1, \ldots, n\}} Q_i \rightarrow^* 0$ for any $Q_1, \ldots, Q_n$. The only situations that prevents $C'$ from progressing are $C' = C[\text{roll\_error}]$ and $C' = C[\text{com\_error}]$. However, from Theorems[9] and [10] respectively, we have $C' \neq C[\text{roll\_error}]$ and $C' \neq C[\text{com\_error}]$, which is a contradiction.