Rollback Recovery in Session-Based Programming

Claudio Antares Mezzina[1(✉)], Francesco Tiezzi[2(✉)], and Nobuko Yoshida[3(✉)]

[1] Università degli Studi di Urbino Carlo Bo, Urbino, Italy
claudio.mezzina@uniurb.it
[2] Università degli Studi di Firenze, Florence, Italy
francesco.tiezzi@unifi.it
[3] University of Oxford, Oxford, UK
nobuko.yoshida@cs.ox.ac.uk

Abstract. To react to unforeseen circumstances or amend abnormal situations 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. Rollbacks in our language always bring the system to previous visited states and 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 in MAUDE. We show that the language satisfies error-freedom and progress of a session.

1 Introduction

Reversible computing [1,26] has gained interest for its application to different fields: from modelling biological/chemical phenomena [18], to simulation [29], debugging [13] and modelling fault-tolerant systems [11,19,32]. Our interest focuses on this latter application and stems from the fact that reversibility can be used to rigorously model, implement and revisit programming abstractions for reliable software systems.

Recent works [4,6,24,25,30] have studied the effect of reversibility in communication-centric scenarios, as a way to correct faulty computations by bringing

This research was funded in whole, or in part, by EPSRC EP/T006544/2, EP/K011715/1, EP/K034413/1, EP/L00058X/1, EP/N027833/2, EP/N028201/1, EP/T014709/2, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS and Horizon EU TaRDIS 101093006. This work was also partially supported by the Italian MUR PRIN 2020 project NiRvAna, the Italian MUR PRIN 2017 project SEDUCE n. 2017TWRCNB, the French ANR project ANR-18-CE25-0007 DCore, the INdAM - GNCS Project "Proprietà qualitative e quantitative di sistemi reversibili" n. CUP E55F2200027001, and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NextGenerationEU. For the purpose of Open Access, the authors have applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.

back the system to a previous consistent 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 of 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]. None of the aforementioned 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. We explain below the distinctive aspects of our checkpoint-based rollback recovery approach.

Linguistic Primitives to Explicitly Program 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., [2–4]) 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` (checkpoint-based rollback recovery pi-calculus), a variant of the session-based π-calculus [17,36] enriched with rollback recovery primitives. We present here a binary version of the calculus, which is more convenient to demonstrate the essence of our rollback recovery approach; the proposed approach can be seamlessly extended to multiparty sessions (see the companion technical report [27] available online). 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 [5]), 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, which are 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 properties that are relaxed and different with respect to other reversible calculi [9,10,21,30]. 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, (b) compliance-checked `cherry-pi` specifications never lead to communication errors (e.g., a blocked communication where there is a receiver without the corresponding sender), and (c) compliance-checked `cherry-pi` specifications 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., [36]), 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 the compliance check using the MAUDE [8] framework (the code is available at https://github.com/tiezzi/cherry-pi).

Outline. Section 2 illustrates the key idea of our rollback recovery approach; Sect. 3 introduces the `cherry-pi` calculus; Sect. 4 introduces typing and compliance checking; Sect. 5 presents the properties satisfied by `cherry-pi`; Sect. 6 concludes the paper with related and future work. Omitted rules, extension to multiparty sessions, proofs of the results, and a further example are reported on the companion technical report [27].

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`.

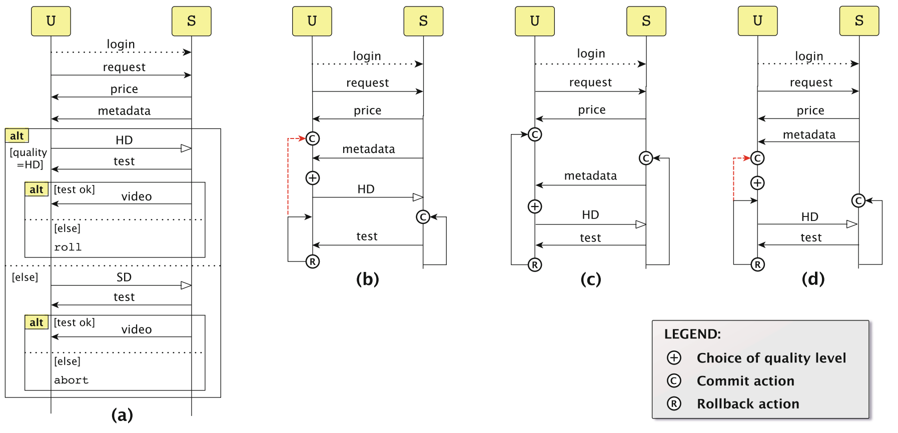

Fig. 1. VOD example: (a) a full description without commit actions; (b, d) runs with undesired rollback; (c) a run with satisfactory rollback.

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 commits. 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 `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.

3 The `cherry-pi` Calculus

In this section, we introduce `cherry-pi`, a calculus (extending that in [36]) devised for studying sessions equipped with our checkpoint-based rollback recovery mechanism.

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, with a slight abuse of notation, 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 `bool`, `int` and `str`, respectively), which are exchanged within a session; *variables* (ranged over by x, y, z), storing values and session endpoints; *process variables* (ranged over by X), used for recursion.

Collaborations, 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.

$C ::=$	**Collaborations**
$\bar{a}(x).P \;\mid\; a(x).P \;\mid\; C_1 \,\vert\, C_2$	request, accept, parallel
$P ::=$	**Processes**
$x!\langle e\rangle.P \;\mid\; x?(y:S).P$	output, input
$\mid\; x \lhd l.P \;\mid\; x \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$	selection, branching
$\mid\; \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \;\mid\; X \;\mid\; \mu X.P \;\mid\; \mathbf{0}$	choice, recursion, inact
$\mid\; \mathsf{commit}.P \;\mid\; \mathsf{roll} \;\mid\; \mathsf{abort}$	commit, roll, abort
$e ::= \; v \;\mid\; +(e_1, e_2) \;\mid\; \wedge(e_1, e_2) \;\mid\; \ldots$	**Expressions**

Fig. 2. `cherry-pi` syntax.

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 $\bar{a}(x).P$ 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 empty process $\mathbf{0}$ and basic actions by means of action prefix $\_ . \_$, conditional choice $\mathsf{if}\ e\ \mathsf{then}\ \_\ \mathsf{else}\ \_$, and recursion $\mu X.\_$. Actions $x!\langle e\rangle$ and $y?(z : S)$ 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 $x \lhd l$ and $x \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$ denote selection and branching (where $l_1, \ldots, l_n$ are pairwise distinct).

Example 1. Let us consider the VOD example informally introduced in Sect. 2. The scenario described in Fig. 1(a) with commit actions placed as in Fig. 1(b) is rendered in `cherry-pi` as $C_{\text{US}} = \overline{login}(x).\, P_{\text{U}} \;\mid\; login(y).\, P_{\text{S}}$, where:

$$\begin{aligned}
P_{\text{U}} \;=\;& x!\langle v_{\mathit{req}}\rangle.\, x?(x_{\mathit{price}} : \mathtt{int}).\, \mathsf{commit}.\, x?(x_{\mathit{meta}} : \mathtt{str}).\, \mathsf{if}\ (f_{\mathit{eval}}(x_{\mathit{price}}, x_{\mathit{meta}}))\\
&\quad \mathsf{then}\ x \lhd l_{\mathit{HD}}.\, x?(x_{\mathit{testHD}} : \mathtt{str}).\\
&\qquad\quad (\mathsf{if}\ (f_{\mathit{HD}}(x_{\mathit{testHD}}))\ \mathsf{then}\ x?(x_{\mathit{videoHD}} : \mathtt{str}).\mathbf{0}\ \mathsf{else}\ \mathsf{roll})\\
&\quad \mathsf{else}\ x \lhd l_{\mathit{SD}}.\, x?(x_{\mathit{testSD}} : \mathtt{str}).\\
&\qquad\quad (\mathsf{if}\ (f_{\mathit{SD}}(x_{\mathit{testSD}}))\ \mathsf{then}\ x?(x_{\mathit{videoSD}} : \mathtt{str}).\mathbf{0}\ \mathsf{else}\ \mathsf{abort})
\end{aligned}$$

$$
\begin{array}{llr}
C ::= \bar{a}(x).P \mid a(x).P \mid C_1 | C_2 \mid (\nu s : C_1)\, C_2 \mid \langle P_1 \rangle \blacktriangleright P_2 & & \textbf{Collaborations} \\
P ::= r\,!\langle e \rangle.P \mid r\,?(y : S).P \mid r \triangleleft l.P \mid r \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \mid \cdots & & \textbf{Processes}
\end{array}
$$

Fig. 3. cherry-pi runtime syntax (the rest of processes P and expressions e are as in Fig. 2).

$$
\begin{aligned}
P_S \;=\; & y?(y_{req} : \texttt{str}).\, y!\langle f_{price}(y_{req})\rangle.\, y!\langle f_{meta}(y_{req})\rangle. \\
& y \rhd \{\, l_{HD} : \mathsf{commit}.\, y!\langle f_{testHD}(y_{req})\rangle.\, y!\langle f_{videoHD}(y_{req})\rangle.\, \mathbf{0}\,, \\
& \qquad l_{SD} : \mathsf{commit}.\, y!\langle f_{testSD}(y_{req})\rangle.\, y!\langle f_{videoSD}(y_{req})\rangle.\, \mathbf{0}\, \}
\end{aligned}
$$

Notice that expressions used for decisions and computations are abstracted by relations $f_n(\cdot)$, whose definitions are left unspecified. Considering the placement of commit actions depicted in Fig. 1(c), the cherry-pi specification of the service's process becomes:

$$
\begin{aligned}
& y?(y_{req} : \texttt{str}).\, y!\langle f_{price}(y_{req})\rangle.\, \mathsf{commit}.\, y!\langle f_{meta}(y_{req})\rangle. \\
& y \rhd \{\, l_{HD} : y!\langle f_{testHD}(y_{req})\rangle.\, y!\langle f_{videoHD}(y_{req})\rangle.\, \mathbf{0}\,, \\
& \qquad l_{SD} : y!\langle f_{testSD}(y_{req})\rangle.\, y!\langle f_{videoSD}(y_{req})\rangle.\, \mathbf{0}\, \}
\end{aligned}
$$

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

$$
x!\langle v_{req}\rangle.\, x?(x_{price} : \texttt{int}).\, x?(x_{meta} : \texttt{str}).\, \mathsf{commit}.\, \mathsf{if}\ (f_{eval}(x_{price}, x_{meta}))\ \mathsf{then}\ \ldots
$$

Semantics. The operational semantics of cherry-pi is defined for *runtime* terms, generated by the extended syntax of the calculus in Fig. 3 (new constructs are highlighted by a grey background). We use r to denote *session identifiers*, i.e. session endpoints and variables. Those runtime terms that can be also generated by the grammar in Fig. 2 are called *initial collaborations*.

At collaboration level, two constructs are introduced: $(\nu s : C_1)\, C_2$ 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; $\langle P_1 \rangle \blacktriangleright P_2$ 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 $(\nu s : C_1)\, C_2$, which binds session endpoints s and $\bar{s}$ in C_2 (in this respect, it acts similarly to the restriction of π-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 correspond 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.

$$k!\langle e\rangle.P \xrightarrow{k!\langle v\rangle} P \quad (e\downarrow v) \ [\text{P-SND}] \qquad\qquad k?(x:S).P \xrightarrow{k?(x)} P \ [\text{P-RCV}]$$

$$k \lhd l.P \xrightarrow{k\lhd l} P \ [\text{P-SEL}] \qquad\qquad k \rhd \{l_1 : P_1, \ldots, l_n : P_n\} \xrightarrow{k\rhd l_i} P_i \ (1 \leqslant i \leqslant n) \ [\text{P-BRN}]$$

$$\mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \xrightarrow{\tau} P_1 \quad (e \downarrow \mathtt{true}) \ [\text{P-IFT}]$$

$$\mathsf{commit}.P \xrightarrow{cmt} P \ [\text{P-CMT}] \qquad\qquad \mathsf{roll} \xrightarrow{roll} \mathbf{0} \ [\text{P-RLL}] \qquad\qquad \mathsf{abort} \xrightarrow{abt} \mathbf{0} \ [\text{P-ABT}]$$

Fig. 4. `cherry-pi` semantics: auxiliary labelled relation.

The operational semantics of `cherry-pi` is given in terms of a standard *structural congruence* $\equiv$ [17] and a *reduction* relation $\rightarrowtail$ given as the union of the *forward reduction* relation $\twoheadrightarrow$ and *backward reduction* relations $\rightsquigarrow$. The definition of the relation $\twoheadrightarrow$ over closed collaborations relies on an auxiliary labelled relation $\xrightarrow{\ell}$ over processes that specifies the actions that processes can initially perform and the continuation process obtained after each such action. 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 in Fig. 4 (omitted rules are in [27]). We use k to denote *generic session endpoints* (s or $\bar{s}$). Action label ℓ stands for either $k!\langle v\rangle$, $k?(x)$, $k \lhd l$, $k \rhd l$, cmt, $roll$, abt, or τ. The meaning of the 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 (auxiliary function $e \downarrow v$ says that closed expression e evaluates to value v).

The operational rules defining the reduction relation $\rightarrowtail$ are reported in Fig. 5 (standard rules for congruence, in the forward and backward case, are omitted). We comment on salient points. Once a session is created, its initiating collaboration is 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]). Notably, this form of commit is asynchronous as it does not require the passive participant to explicitly synchronise with the active participant by means of a primitive for accepting the commit. On the other hand, under the hood, a low-level implementation of this mechanism would synchronously update the logs of the involved parties. Conversely, a rollback action restores the processes in the two logs (rule [B-RLL]). The abort action (rule [B-ABT]), instead, kills the session and restores the collaboration stored in the session construct formed by the two initiators that have started the session; this allows the initiators to be involved in new sessions. The other rules simply extend the standard parallel, restriction rules to forward and backward relations.

Example 2. Consider the first `cherry-pi` specification of the VOD scenario given in Example 1. In the initial state C_{US} of the collaboration, `U` and `S` can synchronise in order to initialise the session, thus evolving to $C^1_{\text{US}} = (\nu s : C_{\text{US}})\ (\langle P_{\text{U}}[\bar{s}/x]\rangle \blacktriangleright P_{\text{U}}[\bar{s}/x] \mid \langle P_{\text{S}}[s/y]\rangle \blacktriangleright P_{\text{S}}[s/y])$.

$$\bar{a}(x_1).P_1 \mid a(x_2).P_2 \twoheadrightarrow (\nu s : (\bar{a}(x_1).P_1 \mid a(x_2).P_2)) \quad \text{[F-CON]}$$
$$(\langle P_1[\bar{s}/x_1]\rangle \blacktriangleright P_1[\bar{s}/x_1] \mid \langle P_2[s/x_2]\rangle \blacktriangleright P_2[s/x_2])$$

$$\frac{P_1 \xrightarrow{\bar{k}!\langle v\rangle} P_1' \qquad P_2 \xrightarrow{k?(x)} P_2'}{\langle Q_1\rangle \blacktriangleright P_1 \mid \langle Q_2\rangle \blacktriangleright P_2 \twoheadrightarrow \langle Q_1\rangle \blacktriangleright P_1' \mid \langle Q_2\rangle \blacktriangleright P_2'[v/x]} \text{[F-COM]} \qquad \frac{C_1 \twoheadrightarrow C_1'}{C_1 \mid C_2 \twoheadrightarrow C_1' \mid C_2} \text{[F-PAR]}$$

$$\frac{P_1 \xrightarrow{\bar{k}\lhd l} P_1' \qquad P_2 \xrightarrow{k\rhd l} P_2'}{\langle Q_1\rangle \blacktriangleright P_1 \mid \langle Q_2\rangle \blacktriangleright P_2 \twoheadrightarrow \langle Q_1\rangle \blacktriangleright P_1' \mid \langle Q_2\rangle \blacktriangleright P_2'} \text{[F-LAB]} \qquad \frac{C_2 \twoheadrightarrow C_2'}{(\nu s : C_1)\, C_2 \twoheadrightarrow (\nu s : C_1)\, C_2'} \text{[F-RES]}$$

$$\frac{P_1 \xrightarrow{cmt} P_1'}{\langle Q_1\rangle \blacktriangleright P_1 \mid \langle Q_2\rangle \blacktriangleright P_2 \twoheadrightarrow \langle P_1'\rangle \blacktriangleright P_1' \mid \langle P_2\rangle \blacktriangleright P_2} \text{[F-CMT]} \qquad \frac{P \xrightarrow{\tau} P'}{\langle Q\rangle \blacktriangleright P \twoheadrightarrow \langle Q\rangle \blacktriangleright P'} \text{[F-IF]}$$

$$\frac{P_1 \xrightarrow{roll} P_1'}{\langle Q_1\rangle \blacktriangleright P_1 \mid \langle Q_2\rangle \blacktriangleright P_2 \rightsquigarrow \langle Q_1\rangle \blacktriangleright Q_1 \mid \langle Q_2\rangle \blacktriangleright Q_2} \text{[B-RLL]} \qquad \frac{C_1 \rightsquigarrow C_1'}{C_1 \mid C_2 \rightsquigarrow C_1' \mid C_2} \text{[B-PAR]}$$

$$\frac{P_1 \xrightarrow{abt} P_1'}{(\nu s : C)(\langle Q_1\rangle \blacktriangleright P_1 \mid \langle Q_2\rangle \blacktriangleright P_2) \rightsquigarrow C} \text{[B-ABT]} \qquad \frac{C_2 \rightsquigarrow C_2'}{(\nu s : C_1) C_2 \rightsquigarrow (\nu s : C_1) C_2'} \text{[B-RES]}$$

Fig. 5. `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:

$$C^2_{\mathrm{US}} = (\nu s : C_{\mathrm{US}})\,(\langle P'_{\mathrm{U}}\rangle \blacktriangleright P'_{\mathrm{U}} \mid \langle P'_{\mathrm{S}}\rangle \blacktriangleright P'_{\mathrm{S}})$$
$$P'_{\mathrm{U}} = \bar{s}?(x_{meta} : \mathtt{str}).\,\mathsf{if}\,(f_{eval}(v_{price}, x_{meta}))\,\mathsf{then}\ldots \quad P'_{\mathrm{S}} = s!\langle f_{meta}(v_{req})\rangle.\, y \rhd \{\ldots\}$$

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

$$\begin{aligned} C^3_{\mathrm{US}} &= (\nu s : C_{\mathrm{US}})\,(\langle P''_{\mathrm{U}}\rangle \blacktriangleright P''_{\mathrm{U}} \mid \langle P''_{\mathrm{S}}\rangle \blacktriangleright P''_{\mathrm{S}}) \\ P''_{\mathrm{U}} &= \bar{s}?(x_{testHD} : \mathtt{str}).\,\mathsf{if}\,(f_{HD}(x_{testHD}))\,\mathsf{then}\,\bar{s}?(x_{videoHD} : \mathtt{str}).\mathbf{0}\,\mathsf{else\ roll} \\ P''_{\mathrm{S}} &= s!\langle f_{testHD}(v_{req})\rangle.\, s!\langle f_{videoHD}(v_{req})\rangle.\,\mathbf{0} \end{aligned}$$

In the next reductions, U evaluates the test video and decides to revert the session execution, resulting in $C^4_{\mathrm{US}} = (\nu s : C_{\mathrm{US}})\,(\langle P''_{\mathrm{U}}\rangle \blacktriangleright \mathsf{roll} \mid \langle P''_{\mathrm{S}}\rangle \blacktriangleright s!\langle f_{videoHD}(v_{req})\rangle.\,\mathbf{0})$. The execution of the roll action restores the checkpoints P''_{U} and P''_{S}, that is $C^4_{\mathrm{US}} \rightsquigarrow C^3_{\mathrm{US}}$. After the rollback, U is not able to change the video quality as her own commit point would have permitted; in fact, it holds $C^4_{\mathrm{US}} \not\rightsquigarrow C^2_{\mathrm{US}}$.

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 Example 2, it does not guarantee high-level properties about the safe execution of the rollback. To prevent such undesired rollbacks,

$$\frac{\emptyset;\emptyset \vdash P \blacktriangleright x : T}{\bar{a}(x).P \ \blacktriangleright \ \{\bar{a} : T\}}\,[\text{T-REQ}] \qquad \frac{\emptyset;\emptyset \vdash P \blacktriangleright x : T}{a(x).P \ \blacktriangleright \ \{a : T\}}\,[\text{T-ACC}] \qquad \frac{C_1 \ \blacktriangleright \ A_1 \quad C_2 \ \blacktriangleright \ A_2}{C_1 \mid C_2 \ \blacktriangleright \ A_1 \cup A_2}\,[\text{T-PAR}]$$

Fig. 6. Typing system for `cherry-pi` collaborations.

we propose the use of *compliance checking*, to be performed at design time. This check is not done on the full system specification, but only at the level of session types.

Session Types and Typing. The syntax of the `cherry-pi` *session types* T is defined as follows. Type $![S].T$ represents the behaviour of first outputting a value of sort S (i.e., `bool`, `int` or `str`), then performing the actions prescribed by type T. Type $?[S].T$ is the dual one, where a value is received instead of sent. Types `end` and `err` represent inaction and faulty termination, respectively. Type $\lhd[l].T$ represents the behaviour that selects the label l and then behaves as T. Type $\rhd[l_1 : T_1, \ldots, l_n : T_n]$ describes a branching behaviour: it waits for one of the n options to be selected, 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. Type `cmt`.T represents a commit action followed by the actions prescribed by type T. Finally, types `roll` and `abt` represent rollback and abort actions.

The `cherry-pi` type system does not perform compliance checks, but only infers the types of collaboration participants, which will be then checked together according to the compliance relation. *Typing judgements* are of the form $C \ \blacktriangleright \ A$, 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, $C \ \blacktriangleright \ A$ indicates that from the collaboration C the type associations in A are inferred. The definition of the type system for these judgements relies on auxiliary typing judgements for processes, of the form $\Theta; \Gamma \ \vdash \ P \ \blacktriangleright \ \Delta$, where Θ, Γ and Δ, 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 $\Theta \cdot X : t$ and $\Gamma \cdot y : S$, where $X \notin dom(\Theta)$, $t \notin cod(\Theta)$ and $y \notin dom(\Gamma)$. The judgement $\Theta; \Gamma \vdash P \ \blacktriangleright \ \Delta$ stands for "under the environment $\Theta; \Gamma$, process P has typing Δ". In its own turn, the typing of processes relies on auxiliary judgments for expressions, of the form $\Gamma \ \vdash \ e \ \blacktriangleright \ S$. The axioms and rules defining the typing system for `cherry-pi` collaborations and processes are given in Figs. 6 and 7; typing rules for expressions are standard (see [27]). 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.

Compliance Checking. To check compliance between pairs of session parties, we consider *type configurations* of the form $(T, T') : \langle \tilde{T_1} \rangle \blacktriangleright T_2 \parallel \langle \tilde{T_3} \rangle \blacktriangleright T_4$, consisting in a pair (T, T') of session types, corresponding to the types of the parties at the initiation

$$\frac{\Gamma \vdash e \blacktriangleright S \quad \Theta;\Gamma \vdash P \blacktriangleright x : T}{\Theta;\Gamma \vdash x!\langle e\rangle.P \blacktriangleright x : ![S].T}\ [\text{T-Snd}] \qquad \frac{\Theta;\Gamma \cdot y : S \vdash P \blacktriangleright x : T}{\Theta;\Gamma \vdash x?(y : S).P \blacktriangleright x : ?[S].T}\ [\text{T-Rcv}]$$

$$\Theta;\Gamma \vdash \mathbf{0} \blacktriangleright x : \mathsf{end}\ [\text{T-Inact}] \qquad \frac{\Gamma \vdash e \blacktriangleright \mathsf{bool} \quad \Theta;\Gamma \vdash P_1 \blacktriangleright x : T_1 \quad \Theta;\Gamma \vdash P_2 \blacktriangleright x : T_2}{\Theta;\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \blacktriangleright x : T_1 \oplus T_2}\ [\text{T-If}]$$

$$\Gamma \cdot x : S \vdash x \blacktriangleright S\ [\text{T-Var}]$$

$$\Theta \cdot X : t;\Gamma \vdash X \blacktriangleright t\ [\text{T-PVar}] \qquad \frac{\Theta \cdot X : t;\Gamma \vdash P \blacktriangleright T}{\Theta;\Gamma \vdash \mu X.P \blacktriangleright \mu t.T}\ [\text{T-Rec}]$$

$$\Theta;\Gamma \vdash \mathsf{roll} \blacktriangleright x : \mathtt{roll}\ [\text{T-Rll}] \qquad \frac{\Theta;\Gamma \vdash P \blacktriangleright x : T}{\Theta;\Gamma \vdash \mathsf{commit}.P \blacktriangleright x : \mathtt{cmt}.T}\ [\text{T-Cmt}]$$

$$\Theta;\Gamma \vdash \mathsf{abort} \blacktriangleright x : \mathtt{abt}\ [\text{T-Abt}]$$

Fig. 7. Typing system for `cherry-pi` processes.

of the session, and in the parallel composition of two pairs $\langle \tilde{T_c} \rangle \blacktriangleright T$, where T is the session type of a party and $\tilde{T_c}$ is 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 $\underline{T}$, representing a checkpoint imposed by the other party. The semantics of type configurations, necessary for the definition of the compliance relation, is given in Fig. 8, where label λ stands for either $![S]$, $?[S]$, $\lhd l$, $\rhd l$, τ, `cmt`, `roll`, or `abt`. We comment on the relevant rules. In case of a commit action, the checkpoints of both parties are updated, and the one of the passive party (i.e., the party that has not performed the commit) is marked as 'imposed' (rule [TS-Cmt]$_1$). 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-Cmt]$_2$), as discussed in Sect. 2 (Fig. 1(c)). In case of a roll action (rule [TS-Rll]$_1$), 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-Abt]$_1$), the configuration goes back to the initial state; this allows the type computation to proceed, in order not to affect the compliance check between the two parties.

On top of the above type semantics, we define the compliance relation, inspired by the relation in [3], and prove its decidability.

Definition 1 (Compliance). *Relation* $\dashv\vdash$ *on configurations is defined as follows:* $(T,T') : \langle \tilde{U_1} \rangle \blacktriangleright T_1 \dashv\vdash \langle \tilde{U_2} \rangle \blacktriangleright T_2$ *holds if for all* U_1', T_1', U_2', T_2' *such that* $(T,T') : \langle \tilde{U_1} \rangle \blacktriangleright T_1 \parallel \langle \tilde{U_2} \rangle \blacktriangleright T_2 \longmapsto^* (T,T') : \langle \tilde{U_1'} \rangle \blacktriangleright T_1' \parallel \langle \tilde{U_2'} \rangle \blacktriangleright T_2' \not\longmapsto$ *we have that* $T_1' = T_2' = \mathsf{end}$. *Two types* T_1 *and* T_2 *are* compliant, *written* $T_1 \dashv\vdash T_2$, *if* $(T_1,T_2) : \langle T_1 \rangle \blacktriangleright T_1 \dashv\vdash \langle T_2 \rangle \blacktriangleright T_2$.

Theorem 1. *Let* T_1 *and* T_2 *be two session types, checking if* $T_1 \dashv\vdash T_2$ *holds is decidable.*

This compliance relation is used to define the notion of *rollback safety*.

Definition 2 (Rollback safety). *Let* C *be an initial collaboration, then* C *is* rollback safe *(shortened* roll-safe*) if* $C \blacktriangleright A$ *and for all pairs* $\bar{a} : T_1$ *and* $a : T_2$ *in* A *we have* $T_1 \dashv\vdash T_2$.

$$\mathtt{cmt}.T \xrightarrow{cmt} T \ \text{[TS-CMT]} \qquad \mathtt{roll} \xrightarrow{roll} \mathtt{end} \ \text{[TS-RLL]} \qquad \mathtt{abt} \xrightarrow{abt} \mathtt{end} \ \text{[TS-ABT]}$$

$$\frac{T_1 \xrightarrow{\tau} T_1'}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1' \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2} \ \text{[TS-TAU]}$$

$$\frac{T_1 \xrightarrow{![S]} T_1' \qquad T_2 \xrightarrow{?[S]} T_2'}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1' \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2'} \ \text{[TS-COM]}$$

$$\frac{T_1 \xrightarrow{cmt} T_1' \qquad \tilde{U}_2 \neq T_2}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle T_1' \rangle \blacktriangleright T_1' \parallel \langle \underline{T_2} \rangle \blacktriangleright T_2} \ \text{[TS-CMT}_1\text{]}$$

$$\frac{T_1 \xrightarrow{cmt} T_1' \qquad \tilde{U}_2 = T_2}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle T_1' \rangle \blacktriangleright T_1' \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2} \ \text{[TS-CMT}_2\text{]}$$

$$\frac{T_1 \xrightarrow{roll} T_1'}{(T,T') : \langle U_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle U_1 \rangle \blacktriangleright U_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright U_2} \ \text{[TS-RLL}_1\text{]}$$

$$\frac{T_1 \xrightarrow{roll} T_1'}{(T,T') : \langle \underline{U_1} \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle \underline{U_1} \rangle \blacktriangleright \mathtt{err} \parallel \langle \tilde{U}_2 \rangle \blacktriangleright \mathtt{err}} \ \text{[TS-RLL}_2\text{]}$$

$$\frac{T_1 \xrightarrow{abt} T_1'}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle T \rangle \blacktriangleright T \parallel \langle T' \rangle \blacktriangleright T'} \ \text{[TS-ABT}_1\text{]}$$

Fig. 8. Semantics of types and type configurations (symmetric rules for configurations are omitted).

Example 3. Let us consider again the VOD example. As expected, the first cherry-pi collaboration defined in Example 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_{S} associated by the type system to the user and the service processes, respectively, are as follows:

$$\begin{aligned} T_{\text{U}} &= ![\mathtt{str}].\,?[\mathtt{int}].\,\mathtt{cmt}.\,?[\mathtt{str}].\,(\lhd[l_{HD}].\,?[\mathtt{str}].\,(\,?[\mathtt{str}].\,\mathtt{end} \oplus \mathtt{roll}\,) \\ &\qquad\qquad\qquad\qquad\qquad \oplus \lhd [l_{SD}].\,?[\mathtt{str}].\,(\,?[\mathtt{str}].\,\mathtt{end} \oplus \mathtt{abt}\,)\,) \\ T_{\text{S}} &= ?[\mathtt{str}].\,![\mathtt{int}].\,![\mathtt{str}].\, \rhd [l_{HD} : \mathtt{cmt}.\,![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}\,, \\ &\qquad\qquad\qquad\qquad\qquad\quad l_{SD} : \mathtt{cmt}.\,![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}] \end{aligned}$$

Thus, the resulting initial configuration is $(T_{\text{U}}, T_{\text{S}}) : \langle T_{\text{U}} \rangle \blacktriangleright T_{\text{U}} \parallel \langle T_{\text{S}} \rangle \blacktriangleright T_{\text{S}}$, which can evolve to the configuration $(T_{\text{U}}, T_{\text{S}}) : \langle \underline{T} \rangle \blacktriangleright \mathtt{roll} \parallel \langle U \rangle \blacktriangleright ![\mathtt{str}].\mathtt{end}$, with $T = ?[\mathtt{str}].\,(?[\mathtt{str}].\,\mathtt{end} \oplus \mathtt{roll})$ and $U = ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}$. This configuration evolves to $(T_{\text{U}}, T_{\text{S}}) : \langle \underline{T} \rangle \blacktriangleright \mathtt{err} \parallel \langle U \rangle \blacktriangleright \mathtt{err}$, which cannot further evolve and is not in a completed state (in fact, type err is different from end), meaning that T_{U} and T_{S} are not compliant.

In the scenario described in Fig. 1(c), instead, the type of the server process is as follows: $T_{\text{S}}' = ?[\mathtt{str}].\,![\mathtt{int}].\,\mathtt{cmt}.\,![\mathtt{str}].\, \rhd [l_{HD} : ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}\,, l_{SD} : ![\mathtt{str}].$

$![\mathtt{str}].\,\mathtt{end}]$ and we have $T_{\mathrm{U}} \dashv\vdash T'_{\mathrm{S}}$. Finally, the types of the processes depicted in Fig. 1(d) are:

$$\begin{aligned} T'_{\mathrm{U}} &= ![\mathtt{str}].\,?[\mathtt{int}].\,?[\mathtt{str}].\,\mathtt{cmt}.\,(\lhd[l_{HD}].\,\ldots \oplus \lhd[l_{SD}].\,\ldots) \\ T''_{\mathrm{S}} &= ?[\mathtt{str}].\,![\mathtt{int}].\,![\mathtt{str}].\,\mathtt{cmt}.\,\rhd[l_{HD} : ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}, l_{SD} : ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}] \end{aligned}$$

and we have $T'_{\mathrm{U}} \not\dashv\vdash T''_{\mathrm{S}}$. Indeed, the corresponding initial configuration can evolve to the configuration $(T'_{\mathrm{U}}, T''_{\mathrm{S}}) : \langle \underline{\lhd[l_{HD}].\,\ldots} \rangle \blacktriangleright \mathtt{roll} \parallel \langle \rhd[l_{HD} : \ldots, l_{SD} : \ldots] \rangle \blacktriangleright ![\mathtt{str}].\mathtt{end}$, which again evolves to a configuration that is not in a completed state.

MAUDE Implementation. To show the feasibility of our approach, we have implemented the semantics of type configurations in Fig. 8 in the MAUDE framework [8]. MAUDE provides an instantiation of rewriting logic [22] and it has been used to implement the semantics of several formal languages [23].

The syntax of `cherry-pi` types and type configurations is specified by defining algebraic data types, while transitions and reductions are rendered as rewrites 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. The CHERRY-TYPES-SYNTAX module 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

```
op roll : -> Type .
```

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

```
op ![_]._ : Sort Type -> Type [frozen prec 25] .
```

To prevent undesired rewrites inside operator arguments, following the approach in [33], we have declared these operations as `frozen`. The `prec` attribute has been used to define the precedence among operators. The CHERRY-TYPES-SEMANTICS module provides *rewrite rules*, and additional operators and equations, to define the `cherry-pi` type semantics. For example, the operational rule [TS-SND] is rendered as follows:

```
rl [TS-Snd] : ![S].T => {![S]}T .
```

The correspondence between the operational rule and the rewrite rule 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 [TS-COM] rule is rendered as:

```
crl [TS-Com] :
  init(T,T') CT1 > T1 || CT2 > T2  =>  init(T,T') CT1 > T1' || CT2 > T2'
  if T1 => {![S]}T1' /\ T2 => {?[S]}T2' .
```

Again, there is a close correspondence between the operational rule and the rewrite one.

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 the configurations reachable from an initial configuration. Specifically, the compliance check between types `T1` and `T2` is rendered as follows:

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

This command searches for all terminal states (`=>!`), i.e. states that cannot be rewritten any more (see $\not\longmapsto$ in Definition 1), and checks if at least one of the two session types in the corresponding configurations (`T1'` and `T2'`) is different from the `end` type. Thus, if this search has no solution, `T1` and `T2` are compliant; otherwise, they are not compliant and a violating configuration is returned.

Example 4. Let us consider the `cherry-pi` types defined in Example 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:

```
eq Tuser = ![str]. ?[int]. cmt. ?[str].
          ((sel['hd]. ?[str]. ((?[str]. end) (+) roll))
           (+) (sel['sd]. ?[str]. ((?[str]. end) (+) abt))) .

eq Tservice = ?[str]. ![int]. ![str].
              brn[brnEl('hd, cmt. ![str]. ![str]. end);
                  brnEl('sd, cmt. ![str]. ![str]. end)] .

eq InitConfig = init(Tuser,Tservice)
                ckp(Tuser) > Tuser || ckp(Tservice) > Tservice .
```

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 checked by loading the two modules of our MAUDE implementation, and 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 --> err
CT2 --> ckp(![str]. ![str]. end)
T2 --> err
```

As explained in Example 3, the two types are not compliant. Indeed, the configuration above is a terminal state, and `T1` and `T2` are clearly different from `end`.

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

```
eq Tservice' = ?[str]. ![int]. cmt. ![str].
               brn[brnEl('hd, ![str]. ![str]. end);
                   brnEl('sd, ![str]. ![str]. end)] .
```

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

```
No solution.
```

meaning that types `Tuser` and `Tservice'` 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 in that case the user and service types are not compliant.

5 Properties of `cherry-pi`

This section presents the results regarding the properties of `cherry-pi`. 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 end, we can instrument the reduction semantics of collaborations by means of labels of the form $cmt\ s$, $roll\ s$ and $abt\ s$, indicating the rule used to derive the reduction and the session on which such operation has been done.

Rollback Properties. We show some properties concerning the reversible behaviour of `cherry-pi` related to the interplay between rollback and commit primitives. The first two properties, namely Theorem 2 and Lemma 1, are an adaptation of typical properties of reversible calculi, while Lemma 2 and Lemma 3 are brand new.

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 mixed executions (also involving 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.

Theorem 2. *Let C_0 be an initial collaboration. If $C_0 \rightarrowtail^* C_1$ then $C_0 \twoheadrightarrow^* C_1$.*

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 `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.

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

A rollback always brings the system to the last taken checkpoint. We recall that, since there may be sessions running in parallel, a collaboration may be able to do different rollbacks within different sessions. Thus, determinism only holds relative to a given session, and rollback within one session has no effect on any other parallel session.

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

$$\frac{P_1 \xrightarrow{k!\langle v\rangle} P_1' \quad \neg P_2 \Downarrow_{k?} \quad \neg P_2 \Downarrow_{roll}}{\langle \tilde{Q}_1\rangle \blacktriangleright P_1 \mid \langle \tilde{Q}_2\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{com\_error}}\ [\text{E-COM1}] \qquad \frac{P_1 \xrightarrow{roll} P_1'}{\langle \underline{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q}_2\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{roll\_error}}\ [\text{E-RLL}_2]$$

Fig. 9. `cherry-pi` semantics: error reductions.

The last rollback property states that a collaboration cannot go back to a state prior to the execution of a commit action, that is commits have a persistent effect. Let us note that recursion does not affect this theorem, since at the beginning of a collaboration computation there is always a new session establishment, leading to a stack of past configurations. Hence it is never the case that from a collaboration C you can reach again C via forward steps.

Theorem 3 (Commit persistency). *Let C be a reachable collaboration. If $C \overset{cmt\ s}{\twoheadrightarrow} C'$ then there exists no C'' such that $C' \twoheadrightarrow^{*} \overset{roll\ s}{\rightsquigarrow} C''$ and $C'' \twoheadrightarrow^{+} C$.*

Soundness Properties. The second group of properties concerns soundness guarantees. The definition of these properties requires formally characterising the errors that may occur in the execution of an unsound collaboration. We rely on error reduction (as in [7]) rather than on the usual static characterisation of errors (as, e.g., in [36]), 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:

$$C ::= \ldots \ \mid\ \langle \tilde{P}_1\rangle \blacktriangleright P_2 \ \mid\ \texttt{roll\_error} \ \mid\ \texttt{com\_error}$$

where $\tilde{P}$ denotes either a checkpoint P committed by the party or a checkpoint $\underline{P}$ imposed by the other party of the session. The semantics of `cherry-pi` is extended as well by the (excerpt) of error reduction rules in Fig. 9. 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 *barb* predicate: $P \Downarrow_{\mu}$ holds if there exists P' such that $P \Rightarrow P'$ and P' can perform an action μ, where μ stands for $k?$, $k!$,$k \lhd l$, $k \rhd l$, or *roll* (i.e., input, output, select, branching action along session channel k, or roll action); $\Rightarrow$ is the reflexive and transitive closure of $\xrightarrow{\tau}$. 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 ready to perform neither the corresponding input nor a roll back (rule $[\text{E-COM}]_1$) or vice versa, or one participant is willing to perform a selection but the corresponding branching is not available on the other side or viceversa. Instead, a *rollback error* takes place in a collaboration when a participant is willing to perform a rollback action but her checkpoint has been imposed by the other participant ($[\text{E-RLL}]_2$). 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: $\mathbb{C} ::= [\cdot] \mid \mathbb{C}|C \mid (\nu s : C)\,\mathbb{C}$.

Definition 3 (Erroneous collaborations). *A collaboration C is* communication (resp. rollback) erroneous *if $C = \mathbb{C}[\mathtt{com\_error}]$ (resp. $C = \mathbb{C}[\mathtt{roll\_error}]$).*

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 roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$.*

Theorem 5 (Communication soundness). *If C is a roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\mathtt{com\_error}]$.*

We conclude with a progress property 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 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 roll-safe collaboration. If $C \rightarrowtail^* C'$ then either $C' \rightarrowtail C''$ for some C'' or $C' \equiv (\nu s : C)(\langle \tilde{Q}_1 \rangle \blacktriangleright \mathbf{0} \mid \langle \tilde{Q}_2 \rangle \blacktriangleright \mathbf{0})$ for some $\tilde{Q}_1$ and $\tilde{Q}_2$.*

6 Conclusion and Related Work

This paper proposes rollback recovery primitives for session-based programming. These primitives come together with session typing, enabling a design time compliance check which ensures checkpoint persistency properties (Lemma 1 and Theorem 3) and session soundness (Theorems 4 and 5). Our compliance check has been implemented in MAUDE.

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 [11,34]. Differently from these works, we have introduced a way to control reversibility by both *triggering* it and *limiting* its scope. Reversibility is triggered via an explicit rollback primitive (as in [20]), while explicit commits limit the scope of potential future reverse executions (as in [11,34]). Differently from [11,34], commit does not require any synchronisation, as it is a local decision. This could lead to run-time 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 rules out such cases.

Reversibility in behavioural types has been studied in different formalisms: *contracts* [2,4], *binary session types* [24], *multiparty session types* [6,25,30,31], and *global graphs* [14,28]. In [2,4] choices can be seen as *implicit* checkpoints and the system can go back to a previous choice and try another branch. In [2] rollback is triggered non-deterministically , while in [4] it is triggered by the system only when the computation is stuck. In both works reversibility (and rollbacks) is used to achieve a relaxed variant of client-server compliance: if there exists an execution in which the client is able to terminate then the client and server are compliant. Hence, reversibility is used as a means to explore different branches if the current one leads to a deadlock. In [24] reversibility is studied in the context of binary session types. Types information is used at run-time by monitors, for binary [24] and multiparty [25] settings, to keep track of the computational history of the system. allowing to revert any computational step. where global types are enriched with computational history. There, reversibility is uncontrolled, and each computational step can be undone. In [6] global types are enriched with history information, and choices are seen as labelled checkpoints. The information about checkpoints is projected into local types. 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. meaning that the former can be programmed while the latter cannot. Checkpoints are not seen as commits, and a rollback can bring the system to a state prior to several checkpoints. In [30] an *uncontrolled* reversible variant of session π-calculus is presented, while [31] studies different notions of reversibility for both binary and multiparty single sessions. In [14,28] 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. to trigger a rollback several conditions and constraints about loops have to be satisfied. Hence, in order to trigger a rollback a runtime condition should be satisfied.

We detach from these works in several ways. Our checkpoint facility is explicit and checkpointing is not relegated to choices: the programmer can decide at any point when to commit. This is because the programmer may be interested in committing, besides choice points, a series of interactions (e.g., to make a payment irreversible). Once a commit is taken, the system cannot revert to a state prior to it. Our rollback is explicit, meaning that it is the programmer who deliberately triggers a rollback. The extension to the multiparty setting is natural and does not rely on a formalism to describe the global view of the system. Our compliance check, which is decidable, resembles those of [2–4], which are defined for different rollback recovery approaches based on implicit checkpoints.

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 [35]. 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. Also, the way our checkpoints are taken resembles the Communication Induced Checkpoints (CIC) approach [12]; we leave as future work a thoughtful comparison between these two mechanisms.

References

1. Aman, B., et al.: Foundations of reversible computation. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) RC 2020. LNCS, vol. 12070, pp. 1–40. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_1
2. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Compliance for reversible client/server interactions. In: BEAT. EPTCS, vol. 162, pp. 35–42 (2014)
3. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., de'Liguoro, U.: Retractable contracts. In: PLACES 2015. EPTCS, vol. 203, pp. 61–72 (2016)
4. Barbanera, F., Lanese, I., de'Liguoro, U.: Retractable and speculative contracts. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 119–137. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_7
5. Barbanera, F., Lanese, I., de'Liguoro, U.: A theory of retractable and speculative contracts. Sci. Comput. Program. **167**, 25–50 (2018)
6. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Concurrent reversible sessions. In: CONCUR. LIPIcs, vol. 85, pp. 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
7. Chen, T., Dezani-Ciancaglini, M., Scalas, A., Yoshida, N.: On the preciseness of subtyping in session types. Logical Methods Comput. Sci. **13**(2) (2017)
8. All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71999-1
9. Cristescu, I., Krivine, J., Varacca, D.: A Compositional Semantics for the Reversible p-Calculus. In: LICS, pp. 388–397. IEEE (2013)
10. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
11. Danos, V., Krivine, J.: Transactions in RCCS. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 398–412. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_31
12. Elnozahy, E.N., Alvisi, L., Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. **34**(3), 375–408 (2002)
13. Engblom, J.: A review of reverse debugging. In: System, Software, SoC and Silicon Debug Conference (S4D), pp. 1–6 (Sept 2012)
14. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible Choreographies via Monitoring in Erlang. In: Bonomi, S., Rivière, E. (eds.) DAIS 2018. LNCS, vol. 10853, pp. 75–92. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93767-0_6
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053567
16. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
17. Kouzapas, D., Yoshida, N.: Globally governed session semantics. Logical Methods Comput. Sci. **10**(4) (2014)

18. Kuhn, S., Ulidowski, I.: Local reversibility in a calculus of covalent bonding. Sci. Comput. Program. **151**, 18–47 (2018)
19. Lanese, I., Lienhardt, M., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Concurrent flexible reversibility. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 370–390. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_21
20. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
21. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversing higher-order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_33
22. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoret. Comput. Sci. **96**(1), 73–155 (1992)
23. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7–8), 721–781 (2012)
24. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebr. Meth. Program. **90**, 2–30 (2017)
25. Mezzina, C.A., Pérez, J.A.: Causal consistency for reversible multiparty protocols. Log. Methods Comput. Sci. 17(4) (2021)
26. Mezzina, C.A., Schlatte, R., Glück, R., Haulund, T., Hoey, J., Holm Cservenka, M., Lanese, I., Mogensen, T.Æ., Siljak, H., Schultz, U.P., Ulidowski, I.: Software and reversible systems: a survey of recent activities. In: Ulidowski, I., Lanese, I., Schultz, U.P., Ferreira, C. (eds.) RC 2020. LNCS, vol. 12070, pp. 41–59. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-47361-7_2
27. Mezzina, C.A., Tiezzi, F., Yoshida, N.: Rollback Recovery in Session-based Programming. Tech. rep., DiSIA, Univ. Firenze (2023). https://github.com/tiezzi/cherry-pi/raw/main/docs/cherry-pi_TR.pdf
28. Mezzina, C.A., Tuosto, E.: Choreographies for automatic recovery. CoRR abs/1705.09525 (2017). https://arxiv.org/abs/1705.09525
29. Perumalla, K.S., Protopopescu, V.A.: Reversible simulations of elastic collisions. ACM Trans. Model. Comput. Simul. **23**(2), 12:1–12:25 (2013)
30. Tiezzi, F., Yoshida, N.: Reversible session-based pi-calculus. J. Log. Algebr. Meth. Program. **84**(5), 684–707 (2015)
31. Tiezzi, F., Yoshida, N.: Reversing single sessions. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 52–69. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_4
32. Vassor, M., Stefani, J.-B.: Checkpoint/rollback vs causally-consistent reversibility. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 286–303. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_20
33. Verdejo, A., Martí-Oliet, N.: Implementing CCS in Maude 2. In: WRLA. ENTCS, vol. 71, pp. 239–257. Elsevier (2002)
34. de Vries, E., Koutavas, V., Hennessy, M.: Communicating transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_39
35. Weikum, G., Schek, H.J.: Concepts and applications of multilevel transactions and open nested transactions. In: Database Transaction Models for Advanced Applications, pp. 515–553. Morgan Kaufmann (1992)
36. Yoshida, N., Vasconcelos, V.T.: Language primitives and type discipline for structured communication-based programming revisited: two systems for higher-order session communication. Electr. Notes Theor. Comp. Sci. **171**(4), 73–93 (2007)