Logical Methods in Computer Science
Volume 21, Issue 1, 2025, pp. 2:1–2:36
https://lmcs.episciences.org/

Submitted Dec. 06, 2023
Published Jan. 10, 2025

CHECKPOINT-BASED ROLLBACK RECOVERY IN SESSION-BASED PROGRAMMING

CLAUDIO ANTARES MEZZINA [a], FRANCESCO TIEZZI [b], AND NOBUKO YOSHIDA [c]

[a] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy

[b] Università degli Studi di Firenze, Italy

[c] University of Oxford, United Kingdom

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 [ACG+20, MSG+20] has gained interest for its application to different fields: from modelling biological/chemical phenomena [KU18], to simulation [PP13], debugging [Eng12, GLM14, LSU22] and modelling fault-tolerant systems [DK05, LLM+13, VS18]. 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 [BLd17, MP17, MP21, CDG17, TY15] have studied the effect of reversibility in communication-centric scenarios, as a way to correct faulty computations by bringing 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

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/Y005244/1, EP/V000462/1, EP/X015955/1, NCSS/EPSRC VeTSS and Horizon EU TaRDIS 101093006, Italian MUR PRIN 2020 project NiRvAna, the Italian MUR PRIN 2022 project DeKLA, by INdAM – GNCS 2024 project *Modelli composizionali per l'analisi di sistemi reversibili distribuiti (MARVEL)*, and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NextGenerationEU.

DOI:10.46298/LMCS-21(1:2)2025

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 [HLV+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., [BDd14, BDLd16, BLd17]) 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 end, we introduce `cherry-pi` (checkpoint-based rollback recovery pi-calculus), a variant of the session-based π-calculus [YV07] enriched with rollback recovery primitives. 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 [BLd18]), 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 [DK04, CKV13, LMS10, TY15].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., [YV07]), 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 [CDE+07] framework (the code is available at `https://github.com/tiezzi/cherry-pi`).

Outline. Section 2 illustrates the key idea of our rollback recovery approach. Section 3 introduces the `cherry-pi` calculus. Section 4 introduces typing and compliance checking. Section 5 illustrates the MAUDE implementation of the compliance checking. Section 6 presents the properties satisfied by `cherry-pi`. Section 7 shows the application of the `cherry-pi` approach to a speculative execution scenario. Section 8 discusses related work. Finally, Section 9 concludes the paper with future work. The appendix reports on the omitted proofs.

This paper is a revised and enhanced version of [MTY23]. In particular:

- Sec. 3 has been extended with rules in Fig. 4 and 5, which are omitted in [MTY23], to provide a complete account of the `cherry-pi` semantics.
- Sec. 4 has been extended with omitted rules (Fig. 7 and 10 are new, while Fig. 9 and 11 have been extended), to provide a full account of the `cherry-pi` typing discipline. Moreover, the section includes the proof of Theorem 4.2.
- Sec. 6 has been extended by including full proofs of the results regarding the properties of `cherry-pi`.
- Sec. 7 is new. It shows the `cherry-pi` approach at work on a new scenario to provide a better understanding of the practical application of `cherry-pi` and, in particular, of the MAUDE implementation of its type semantics.
- Sec. 8 has been revised and expanded, including the discussion of more recent related work and a table (Tab. 1) providing a comparison of the related approaches in the literature.
- Finally, more commentary and explanations have been added throughout the paper, and the whole presentation has been carefully refined.

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`

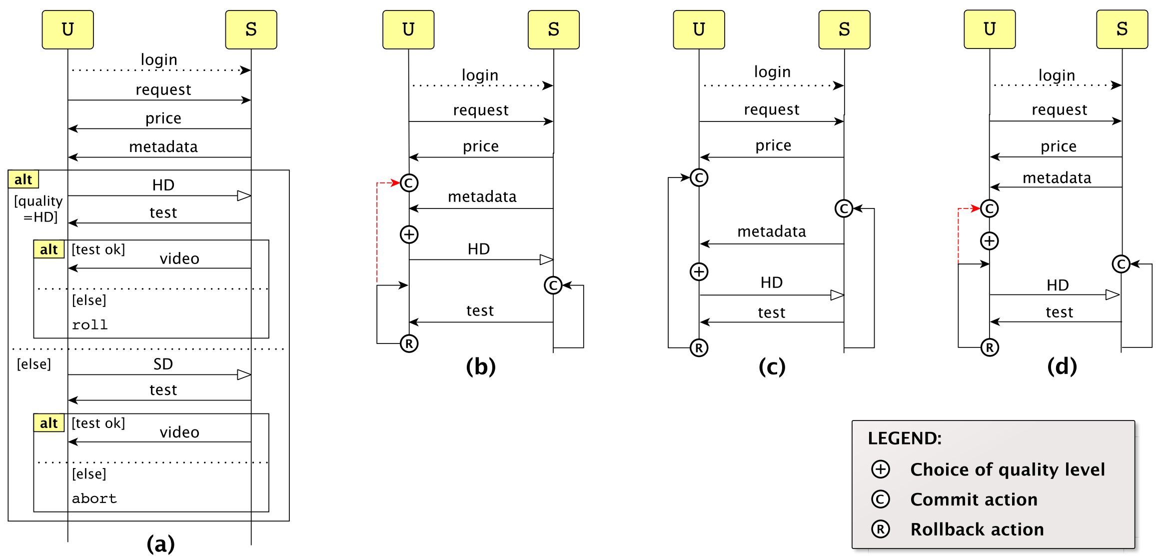

Figure 1: VOD example: (a) a full description without commit actions; (b,d) runs with undesired rollback; (c) a run with satisfactory rollback.

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`. Hence, `U` cannot use the rollback mechanism to undo her video quality choice and select the SD video.

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. Notably, our aim is to provide programming constructs and underlying mechanisms that would be easy to use and understand for the user. Thus, although it would be possible to define more expressive constructs, such as commit and rollback actions specifying a label as a parameter (in a way similar to the approaches introduced in [LMSS11, GLMT17]), we preferred to avoid them as they would make the system's behavior more intricate. In fact, such labelled actions would behave as a sort of `goto` jump, which can lead to unmanageable spaghetti code.

$C ::=$		**Collaborations**
	$\bar{a}(x).P \mid a(x).P \mid C_1 \mid C_2$	request, accept, parallel
$P ::=$		**Processes**
	$x!\langle e\rangle.P \mid x?(y:S).P$	output, input
	$\mid x \triangleleft l.P \mid x \triangleright \{l_1 : P_1, \ldots, l_n : P_n\}$	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**

Figure 2: `cherry-pi` syntax.

3. The `cherry-pi` calculus

In this section, we introduce `cherry-pi`, a calculus (extending that in [YV07]) 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, 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. 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 [HVK98]. A `cherry-pi` collaboration is a collection (more specifically, a parallel composition) 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 $\mathbf{0}$ (which can do nothing) and basic actions by means of action prefix $\_.\_$ (which allows the process on the right of the . operator to proceed once the action on the left of the . operator is executed), conditional choice $\mathsf{if}\ e\ \mathsf{then}\ \_\ \mathsf{else}\ \_$ (which has the usual meaning), and recursion $\mu X.\_$ (which behaves as its process argument where the occurrences of the process variable X are replaced by the recursion process itself). 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[1]. Actions $x \lhd l$ and $x \rhd \{l_1 : P_1, \ldots, l_n : P_n\}$ denote selection and branching respectively (where $l_1, \ldots, l_n$ are pairwise distinct).

Example 3.1. 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 $C_{\mathtt{US}} = \overline{login}(x).\, P_{\mathtt{U}} \mid login(y).\, P_{\mathtt{S}}$, where:

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

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

Notice that expressions used for decisions and computations are abstracted by relations $f_{\mathsf{n}}(\cdot)$, 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:

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

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

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

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 (new constructs are highlighted by a grey background). We use k to denote *generic session endpoints*, i.e. s or $\bar{s}$, and 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.

Bindings are defined as follows: $\bar{a}(x).P$, $a(x).P$, and $r?(x : S).P$ bind variable x in P; $(\nu s : C_1)\ C_2$ 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/extruded to avoid involving

[1]Notably, even if session delegation is not supported, we cannot just consider single binary sessions avoiding the notion of collaborations. In fact, collaborations allows us to consider non-determinism at the level of session establishment (e.g., think of a client and two servers providing the same service).

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

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

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

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

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

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

$$\mathsf{commit}.P \xrightarrow{cmt} P \quad [\text{P-Cmt}] \qquad \mathsf{roll} \xrightarrow{roll} \mathbf{0} \quad [\text{P-Rll}] \qquad \mathsf{abort} \xrightarrow{abt} \mathbf{0} \quad [\text{P-Abt}]$$

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

processes that are not part of the session in the rollback effect); and $\mu X.P$ binds process variable X in P. The occurrence of a name (where name stand for variable, process variable and session endpoint) is *free* if it is not bound; we assume that bound names are pairwise distinct. Two terms are *alpha-equivalent* if one can be obtained from the other by consistently renaming bound names; as usual, we identify terms up to alpha-equivalence. Communication gives rise to *substitutions* of variables with values: we denote with $P[v/x]$ the process obtained by replacing each free occurrence of the variable x in P by the value v. Similarly, $P[Q/X]$ (resp. $P[k/x]$) denotes the process obtained by replacing each free occurrence of X (resp. x) in P by the process Q (resp. generic session identifier k). 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.

The operational semantics of `cherry-pi` is given in terms of a standard *structural congruence* $\equiv$ (given in Fig. 5) and a *reduction* relation $\rightarrowtail$ given as the union of the *forward reduction* relation $\twoheadrightarrow$ and *backward reduction* relation $\rightsquigarrow$. 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. We consider all reduction relations closed under structural congruence. 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. 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).

$$C_1 \mid C_2 \equiv C_2 \mid C_1 \qquad (C_1 \mid C_2) \mid C_3 \equiv C_1 \mid (C_2 \mid C_3) \qquad \mu X.P \equiv P[\mu X.P/X]$$

Figure 5: Structural congruence for `cherry-pi`

The operational rules defining the reduction relation $\rightarrowtail$ are reported in Fig. 6. We comment on salient points. Once a session is created (via a synchronisation along a shared channel a), its initiating collaboration is stored in the session construct (rule [F-Con]); note how a fresh session channel s is generated as the result of the interaction. Communication, branching selection and internal conditional choice proceed as usual, without affecting logs. Specifically, communication takes place when an output and an input action synchronise along a session channel (rule [F-Com]); the delivery of the value v (resulting from the evaluation of the expression argument of the output action, rule [P-Snd]) is expressed by the application of a substitution $[v/x]$ to the continuation of the receiving process. Branching selection results from the synchronisation on a label l (rule [F-Lab]); note that only one of the branches is selected while the remaining ones are discarded (rule [P-Brn]). Internal conditional choice behaves in a standard way (rule [F-If]); as usual, the choice depends by the positive (rule [P-IfT]) or negative (rule [P-IfF]) evaluation of the boolean expression argument of the conditional choice construct. 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 and restriction rules to forward and backward relations.

Example 3.2. Consider the first `cherry-pi` specification of the VOD scenario given in Ex. 3.1. In the initial state $C_{\mathtt{US}}$ of the collaboration, `U` and `S` can synchronise in order to initialise the session, thus evolving to $C^1_{\mathtt{US}} = \big(\nu s : C_{\mathtt{US}}\big)\ (\langle P_{\mathtt{U}}[\bar{s}/x]\rangle \blacktriangleright P_{\mathtt{U}}[\bar{s}/x] \ \mid\ \langle P_{\mathtt{S}}[s/y]\rangle \blacktriangleright P_{\mathtt{S}}[s/y])$.

Let us consider now a possible run of the session. After three reduction steps, `U` executes the commit action, obtaining the following runtime term:

$$C^2_{\mathtt{US}} = \big(\nu s : C_{\mathtt{US}}\big)\ (\langle P'_{\mathtt{U}}\rangle \blacktriangleright P'_{\mathtt{U}} \ \mid\ \langle P'_{\mathtt{S}}\rangle \blacktriangleright P'_{\mathtt{S}})$$

$$P'_{\mathtt{U}} = \bar{s}?(x_{\mathit{meta}} : \mathtt{str}).\,\mathsf{if}\ (f_{\mathit{eval}}(v_{\mathit{price}}, x_{\mathit{meta}}))\ \mathsf{then}\ \ldots \qquad P'_{\mathtt{S}} = s!\langle f_{\mathit{meta}}(v_{\mathit{req}})\rangle.\, y \rhd \{\ \ldots\ \}$$

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

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

In the next reductions, `U` evaluates the test video and decides to revert the session execution, resulting in $C^4_{\mathtt{US}} = \big(\nu s : C_{\mathtt{US}}\big)\ (\langle P''_{\mathtt{U}}\rangle \blacktriangleright \mathsf{roll} \ \mid\ \langle P''_{\mathtt{S}}\rangle \blacktriangleright s!\langle f_{\mathit{videoHD}}(v_{\mathit{req}})\rangle.\,\mathbf{0})$. The execution of the roll action restores the checkpoints $P''_{\mathtt{U}}$ and $P''_{\mathtt{S}}$, that is $C^4_{\mathtt{US}} \rightsquigarrow C^3_{\mathtt{US}}$. After the rollback,

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

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

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

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

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

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

Figure 6: `cherry-pi` semantics: forward and backward reduction relations.

`U` is not able to change the video quality as her own commit point would have permitted; in fact, it holds $C^4_{\texttt{US}} \not\rightsquigarrow C^2_{\texttt{US}}$.

It is worth noticing that our approach does not guarantee the avoidance of infinite loops due to taking the same decisions after rollbacks. However, `cherry-pi` allows the programmer to specify appropriate conditions to exit from these loops. More specifically, when a checkpoint is restored, the state of the interaction protocol is reverted at the committed configuration and, hence, the variables substituted by the undone interactions (see rule [F-COM] in Fig. 6) are restored as well. Anyway, internal choice decisions result from the evaluation of expressions that predicate not only on these (protocol) variables, but also on information concerning the system state and decisions taken by external actors (e.g., humans) that is not subject to the reversibility effect. Relying on this kind of information is possible to make a decision to exit from a loop. These decisions are abstracted in our example scenarios by means of relations whose definitions are left unspecified. For example, in the VOD scenario, the selection of the video quality (HD vs. SD) taken by the user depends on video's price and metadata (stored in the corresponding protocol variables) but also on the budget and the personal opinion of the user herself; on the whole, the selection decision is abstracted by means of the relation $f_{\mathit{eval}}()$, which will return a different result after the rollback even if invoked with the same input data. In fact, despite the interaction protocol has been reverted, the user remembers that the HD video quality was not a good choice. There are different ways to keep track of information of a reverted computation. One could use alternatives [LLM+13] or an external oracle [Vas21]. The management of the information outside the one stored in the protocol variables is out of the scope of this work; we leave this for future investigation.

$$S ::= \texttt{bool} \quad | \quad \texttt{int} \quad | \quad \texttt{str} \qquad\qquad \textbf{Sorts}$$

$$T ::= ![S].T \quad | \quad ?[S].T \quad | \quad \lhd[l].T \quad | \quad \rhd[l_1 : T_1, \ldots, l_n : T_n] \qquad\qquad \textbf{Session types}$$
$$| \quad T_1 \oplus T_2 \quad | \quad t \quad | \quad \mu t.T \quad | \quad \texttt{end} \quad | \quad \texttt{err} \quad | \quad \texttt{cmt}.T \quad | \quad \texttt{roll} \quad | \quad \texttt{abt}$$

Figure 7: `cherry-pi` type syntax.

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. 3.2, 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. This check is not done on the full system specification, but only at the level of session types. In this way, we abstract as sorts the values exchanged via communication actions at process level. This not only permits formulating the compliance check more simply but, even more importantly, it ensures that the semantics produces finite LTSs, thus making the compliance check decidable.

4.1. **Session types and typing.** The syntax of the `cherry-pi` *session types* T is defined in Fig. 7. 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. 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 that starts by doing T and, when variable t is encountered, recurs to T again. Types `end` and `err` represent inaction and faulty termination, respectively. Type $\texttt{cmt}.T$ represents a commit action followed by the actions prescribed by type T. Finally, types `roll` and `abt` represent rollback and abort actions. Notably, `cherry-pi` session types are defined as in [YV07], but session delegation (i.e., channel-passing) is not supported while there are additional types corresponding to the reversibility actions. Concerning choice, the type discipline, as in [YV07], supports internal choice (corresponding to the if-then-else construct at process level) and external choice (corresponding to a label synchronization via interaction between selection and branching constructs at process level). Therefore, no other forms of choice are supported and, differently from other works (e.g., [BDd14, BDLd16, BLd17]), choices are not implicit checkpoints and rollback is explicitly triggered.

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 $X \notin dom(\Theta)$, $t \notin cod(\Theta)$

$$\frac{\varnothing;\varnothing \vdash P \blacktriangleright x : T}{\bar{a}(x).P \;\blacktriangleright\; \{\bar{a} : T\}}[\text{T-Req}] \qquad \frac{\varnothing;\varnothing \vdash P \blacktriangleright x : T}{a(x).P \;\blacktriangleright\; \{a : T\}}[\text{T-Acc}] \qquad \frac{C_1 \;\blacktriangleright\; A_1 \quad C_2 \;\blacktriangleright\; A_2}{C_1 \mid C_2 \;\blacktriangleright\; A_1 \cup A_2}[\text{T-Par}]$$

Figure 8: Typing system for `cherry-pi` collaborations.

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, processes and expressions are given in Fig. 8, 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.

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

$$\Theta;\Gamma \vdash \mathbf{0} \blacktriangleright x : \texttt{end}\ [\text{T-Inact}] \qquad \frac{\Theta;\Gamma \vdash P \blacktriangleright x : T}{\Theta;\Gamma \vdash x \lhd l.P \blacktriangleright x : \lhd[l].T}\ [\text{T-Sel}]$$

$$\frac{\Theta;\Gamma \vdash P_1 \blacktriangleright x : T_1 \quad \dots \quad \Theta;\Gamma \vdash P_n \blacktriangleright x : T_n}{\Theta;\Gamma \vdash x \rhd \{l_1 : P_1, \dots, l_n : P_n\} \blacktriangleright x : \rhd[l_1 : T_1, \dots, l_n : T_n]}\ [\text{T-Br}]$$

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

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

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

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

Figure 9: Typing system for `cherry-pi` processes.

4.2. **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 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

$$\Gamma \vdash \texttt{true} \blacktriangleright \texttt{bool} \ [\text{T-Bool}_{tt}] \qquad \Gamma \vdash \texttt{false} \blacktriangleright \texttt{bool} \ [\text{T-Bool}_{ff}]$$

$$\Gamma \cdot x : S \vdash x \blacktriangleright S \ [\text{T-Var}] \qquad \Gamma \vdash 1 \blacktriangleright \texttt{int} \ [\text{T-Int}] \qquad \Gamma \vdash \text{“}a\text{”} \blacktriangleright \texttt{str} \ [\text{T-Str}]$$

$$\frac{\Gamma \vdash e_1 \blacktriangleright \texttt{int} \quad \Gamma \vdash e_2 \blacktriangleright \texttt{int}}{\Gamma \vdash +(e_1, e_2) \blacktriangleright \texttt{int}} \ [\text{T-Sum}] \qquad \frac{\Gamma \vdash e_1 \blacktriangleright \texttt{bool} \quad \Gamma \vdash e_2 \blacktriangleright \texttt{bool}}{\Gamma \vdash \wedge(e_1, e_2) \blacktriangleright \texttt{bool}} \ [\text{T-And}]$$

Figure 10: Typing system for `cherry-pi` expressions (excerpt of rules).

compliance relation, is given in Fig. 11, 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 Sec. 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; otherwise, the configuration cannot proceed with the rollback and reduces to an erroneous configuration (rule [TS-Rll$_2$]). 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 [BDLd16], and prove its decidability.

Definition 4.1 (Compliance)**.** Relation $\dashV$ on configurations is defined as follows: $(T, T') : \langle \tilde{U_1} \rangle \blacktriangleright T_1 \dashV \langle \tilde{U_2} \rangle \blacktriangleright T_2$ 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 $T_1' = T_2' = \texttt{end}$. Two types T_1 and T_2 are *compliant*, written $T_1 \dashV T_2$, if $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \dashV \langle T_2 \rangle \blacktriangleright T_2$.

Theorem 4.2. *Let T_1 and T_2 be two session types, checking if $T_1 \dashV T_2$ holds is decidable.*

Proof. By Def. 4.1, checking $T_1 \dashV T_2$ consists in checking that types T_1' and T_2' of each configuration $(T_1, T_2) : \langle \tilde{U_1'} \rangle \blacktriangleright T_1' \parallel \langle \tilde{U_2'} \rangle \blacktriangleright T_2'$ such that $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2 \longmapsto^*$ $(T_1, T_2) : \langle \tilde{U_1'} \rangle \blacktriangleright T_1' \parallel \langle \tilde{U_2'} \rangle \blacktriangleright T_2' \not\longmapsto$ (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 $t = (T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2$ by the reduction semantics of types (Fig. 11): the set $\mathcal{S}$ of states corresponds to the set of type configurations reachable from t, i.e. $\mathcal{S} = \{ t' \mid t \longmapsto^* t' \}$, while the set $\mathcal{R}$ of system transitions corresponds to set of the type reductions involving configurations in $\mathcal{S}$, i.e. $\mathcal{R} = \{(t', t'') \in \mathcal{S} \times \mathcal{S} \mid t' \longmapsto t''\}$. Hence, checking $T_1 \dashV T_2$ 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 $(T_1, T_2) : \langle \tilde{V_1} \rangle \blacktriangleright V_2 \parallel \langle \tilde{V_3} \rangle \blacktriangleright V_4$, we have to check if $V_2 = V_4 = \texttt{end}$. 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 6.5), 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 [Mil89, Sec. 7.5]). □

$$!\![S].T \xrightarrow{![S]} T \;\; [\text{TS-SND}] \qquad ?[S].T \xrightarrow{?[S]} T \;\; [\text{TS-RCV}] \qquad \lhd[l].T \xrightarrow{\lhd l} T \;\; [\text{TS-SEL}]$$

$$\rhd[l_1 : T_1, \ldots, l_n : T_n] \xrightarrow{\rhd l_i} T_i \quad (1 \leqslant i \leqslant n) \;\; [\text{TS-BR}] \qquad \frac{T[\mu t.T/t] \xrightarrow{\lambda} T'}{\mu t.T \xrightarrow{\lambda} T'} \; [\text{TS-REC}]$$

$$T_1 \oplus T_2 \xrightarrow{\tau} T_1 \;\; [\text{TS-IF}_1] \qquad T_1 \oplus T_2 \xrightarrow{\tau} T_2 \;\; [\text{TS-IF}_2]$$

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

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

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

$$\frac{T_1 \xrightarrow{\lhd l} T_1' \qquad T_2 \xrightarrow{\rhd l} T_2'}{(T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1 \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2 \longmapsto (T,T') : \langle \tilde{U}_1 \rangle \blacktriangleright T_1' \parallel \langle \tilde{U}_2 \rangle \blacktriangleright T_2'} \; [\text{TS-LAB}]$$

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

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

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

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

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

Figure 11: Semantics of types and type configurations (symmetric rules for configurations are omitted).

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

Definition 4.3 (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 $T_1 \dashv\!\!\vert\; T_2$.

Example 4.4. Let us consider again the VOD example. As expected, the first `cherry-pi` collaboration defined in Ex. 3.1, 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 $T_{\mathtt{U}}$ and $T_{\mathtt{S}}$ associated by the type system to the user and the service processes, respectively, are as follows:

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

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

In the scenario described in Fig. 1(c), instead, the type of the server process is as follows: $T'_{\mathtt{S}} = ?[\mathtt{str}].\,![\mathtt{int}].\,\mathtt{cmt}.\,![\mathtt{str}].\,\rhd[l_{HD}\,: ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}\,,\, l_{SD}\,: ![\mathtt{str}].\,![\mathtt{str}].\,\mathtt{end}]$ and we have $T_{\mathtt{U}} \dashv\!\vdash T'_{\mathtt{S}}$. Finally, the types of the processes depicted in Fig. 1(d) are:

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

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

5. MAUDE implementation.

To show the feasibility of our approach, we have implemented the semantics of type configurations in Fig. 11 in the MAUDE framework [CDE+07]. MAUDE provides an instantiation of rewriting logic [Mes92] and it has been used to implement the semantics of several formal languages [Mes12].

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 [VM02], 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 Def. 4.1), 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 5.1. Let us consider the `cherry-pi` types defined in Ex. 4.4 for the scenario described in Fig. 1(b). In our MAUDE implementation of the type syntax, the session types $T_{\mathtt{U}}$ and $T_{\mathtt{S}}$, 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 Ex. 4.4, 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.

6. 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. When we do not want to distinguish the applied rule we will write $C \overset{s}{\rightarrowtail} C'$ to indicate that a reduction is taking plase on session s.

6.1. Useful lemmata.

We now introduce a couple of results that will allow us to focus on one session per time, even if there could be multiple concurrent sessions running together. The first of such results tells us that reductions taking place on different sessions can be swapped. Formally:

Lemma 6.1 (Swap Lemma)**.** *Let C be a collaboration and s and r two sessions. If $C \overset{s}{\rightarrowtail} C_1 \overset{r}{\rightarrowtail} C_2$ then there exists a collaboration C_3 such that $C \overset{r}{\rightarrowtail} C_3 \overset{s}{\rightarrowtail} C_2$.*

Proof. By case analysis on the reductions $\overset{s}{\rightarrowtail}$ and $\overset{r}{\rightarrowtail}$. □

All the reductions taking place on a session s can be put all together in sequence. Formally:

Lemma 6.2. *Let C be a collaboration. If $C \rightarrowtail^* C_1$, then for any session s in C_1 there exists a collaboration C_0 such that $C \rightarrowtail^* C_0 \overset{s}{\rightarrowtail}^* C_1$ and s is never used in the trace $C \rightarrowtail^* C_0$.*

Proof. By induction on the number n of reduction on s. If there are no reductions then the thesis trivially holds. Otherwise, we can take the very last reduction on s, that is the closest one to C_1 and iteratively apply Lemma 6.1 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 6.2 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 $C \rightarrowtail^* C_1 \overset{s}{\rightarrowtail}^*$ and $s \notin \rightarrowtail^*$, we indicate C_1 as the *initial* collaboration for s. This will allow us to focus just on a sigle session, say s, and to consider collaboration initial for s without loosing of generality.

6.2. Rollback properties.

We show some properties concerning the reversible behaviour of `cherry-pi` related to the interplay between rollback and commit primitives. Notably, Theorem 6.5 and Lemma 6.6, are an adaptation of typical properties of reversible calculi, while Lemma 6.7 and Lemma 6.8 are brand new.

The following two lemmata are the key ingredients for proving Theorem 6.5. Specifically, the former lemma states that an abort leads back to the initial collaboration, while the latter one states that a rollback leads back to the last committed checkpoint.

Lemma 6.3. *Let C be an initial collaboration such that $C \rightarrowtail^* C_1$. If $C_1 \overset{abt}{\rightsquigarrow} C_2$ 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 $C \rightarrowtail^* C_1$ has to be an application of rule [F-Con], that is

$$\begin{aligned} C \twoheadrightarrow & \big(\nu s : (\bar{a}(x_1).P_1 \mid a(x_2).P_2)\big) \\ & (\langle P_1[\bar{s}/x_1]\rangle \blacktriangleright P_1[\bar{s}/x_1] \ \mid \ \langle P_2[s/x_2]\rangle \blacktriangleright P_2[s/x_2]) = C' \end{aligned}$$

and, by hypothesis, $C' \rightarrowtail^* C_1$. Now, no matter the shape of processes in C_1, by applying rule [B-Abt], we will go back to C, that is $C_1 \overset{abt}{\rightsquigarrow} C$, as desired. □

Lemma 6.4. *Let C be a reachable collaboration, such that $C \overset{cmt}{\twoheadrightarrow} C_1$. If $C_1 \twoheadrightarrow^* C_2 \overset{roll}{\rightsquigarrow} C_3$ and there is no commit in $C_1 \twoheadrightarrow^* C_2$, 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 6.3's proof, we can assume $C \equiv \big(\nu s : C_0\big)(\langle P_1\rangle \blacktriangleright P_2 \mid \langle Q_1\rangle \blacktriangleright Q_2)$. Therefore, assuming w.l.o.g. that P_2 is the committing process (i.e., $P_2 \xrightarrow{cmt} P_2'$), we have that $C_1 = \big(\nu s : C_0\big)(\langle P_2'\rangle \blacktriangleright P_2' \mid \langle Q_2\rangle \blacktriangleright Q_2)$. By hypothesis, there is no commits in $C_1 \twoheadrightarrow^* C_2$, and this implies that the log part of C_1 will never change. Hence, we have that $C_2 \equiv \big(\nu s : C_0\big)(\langle P_2'\rangle \blacktriangleright P \mid \langle Q_2\rangle \blacktriangleright Q)$ for some processes P and Q. By applying [B-Rll] we have that $C_2 \overset{roll}{\rightsquigarrow} \big(\nu s : C_0\big)(\langle P_2'\rangle \blacktriangleright P_2' \mid \langle Q_2\rangle \blacktriangleright Q_2) \equiv C_1$, as desired. □

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

Proof. By induction on the number n of backward reductions contained into $C_0 \rightarrowtail^* C_1$. 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:

$$C_0 \twoheadrightarrow^* C' \rightsquigarrow C'' \rightarrowtail^* C_1$$

Depending whether it is an $\overset{abt}{\rightsquigarrow}$ or a $\overset{roll}{\rightsquigarrow}$ we can apply respectively Lemma 6.3 or Lemma 6.4 to obtain a forward trace of the form

$$C_0 \twoheadrightarrow^* C'' \rightarrowtail^* C_1$$

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

We now show a variant of the so-called Loop Lemma [DK04]. 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 6.6 (Safe rollback)**.** *Let C_1 and C_2 be reachable collaborations. If $C_1 \rightsquigarrow C_2$ then $C_2 \twoheadrightarrow^* C_1$.*

Proof. Since C_1 is a reachable collaboration, we have that there exists an initial collaboration C_0 such that $C_0 \rightarrowtail^* C_1$. By applying Theorem 6.5 we can rearrange the trace such that it contains just forward transitions as follows

$$C_0 \twoheadrightarrow^* C_1$$

If the backward reduction is obtained by applying [B-Abt], by Lemma 6.3 we have $C_2 \equiv C_0$, from which the thesis trivially follows. Instead, if the backward reduction is obtained by applying [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:

$$C_0 \twoheadrightarrow^* \overset{cmt}{\twoheadrightarrow} C_{cmt} \twoheadrightarrow^* C_1 \overset{roll}{\rightsquigarrow} C_2$$

and by applying Lemma 6.4 we have that $C_2 \twoheadrightarrow^* \equiv C_1$ as desired.

In the case there is no commit in the trace, without losing of generality we can assume $C_0 \equiv \bar{a}(x_1).P_1 \mid a(x_2).P_2$ and we have:

$$C_0 \twoheadrightarrow \big(\nu s : C_0\big)(\langle P_1' \rangle \blacktriangleright P_1' \mid \langle P_2' \rangle \blacktriangleright P_2') \twoheadrightarrow^* \big(\nu s : C_0\big)(\langle P_1' \rangle \blacktriangleright P_1'' \mid \langle P_2'' \rangle \blacktriangleright P_2') \equiv C_1$$

By rule [B-Rll], we have $C_2 \equiv \big(\nu s : C_0\big)(\langle P_1' \rangle \blacktriangleright P_1' \mid \langle P_2' \rangle \blacktriangleright P_2')$. Thus, we can conclude by noticing that $C_0 \twoheadrightarrow C_2$ is the first reduction in $C_0 \twoheadrightarrow^* C_1$. $\square$

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

Proof. Since C is a reachable collaboration, it is has been generated by an initial collaboration C_0 of the form $C_0 = \bar{a}(x).P_1 \mid a(x).P_2$, and by Theorem 6.5 we have that $C_0 \twoheadrightarrow^* C$. 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:

$$C_0 \twoheadrightarrow^* C_{cmt} \twoheadrightarrow^* C$$

so that in the reduction $C_{cmt} \twoheadrightarrow^* C$ there is no commit. If from C the rollbacks $C \overset{roll}{\rightsquigarrow} C'$ and $C \overset{roll}{\rightsquigarrow} C''$ are triggered by the same process, the thesis trivially follows. In the other case, we have that:

$$C \equiv \big(\nu s : C_0\big)(\langle Q_1\rangle \blacktriangleright Q_1' \mid \langle Q_2\rangle \blacktriangleright Q_2')$$

with both Q_1' and Q_2' able to trigger a rollback. If the roll action is executed by Q_1', by applying rule [B-Rll] we have that

$$\big(\nu s : C_0\big)(\langle Q_1\rangle \blacktriangleright Q_1' \mid \langle Q_2\rangle \blacktriangleright Q_2') \overset{roll}{\rightsquigarrow} \big(\nu s : C_0\big)(\langle Q_1\rangle \blacktriangleright Q_1 \mid \langle Q_2\rangle \blacktriangleright Q_2) \equiv C'$$

If the roll is triggered by Q_2', , by applying rule [B-Rll] up to structural congruence we have that

$$\big(\nu s : C_0\big)(\langle Q_1\rangle \blacktriangleright Q_1' \mid \langle Q_2\rangle \blacktriangleright Q_2') \overset{roll}{\rightsquigarrow} \big(\nu s : C_0\big)(\langle Q_1\rangle \blacktriangleright Q_1 \mid \langle Q_2\rangle \blacktriangleright Q_2) \equiv C''$$

We can conclude by noticing that $C' \equiv C''$, as desired. □

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

Proof. We proceed by contradiction. Suppose that there exists C'' such that $C' \twoheadrightarrow^* \overset{roll}{\rightsquigarrow} C''$ and $C'' \twoheadrightarrow^+ C$. Since C is a reachable collaboration, thanks to Theorem 6.5 we have that there exists an initial collaboration C_0 such that $C_0 \twoheadrightarrow^* C \overset{cmt\,s}{\twoheadrightarrow} C'$. Since a rollback brings back the collaboration to a point before a commit, this means it has been restored a checkpoint committed before the last one (by [B-Rll], indeed, only processes stored in logs can be restored). This implies that there exist at least two different commits in the trace such that

$$C_0 \twoheadrightarrow^* \overset{cmt}{\twoheadrightarrow} C_{cmt} \twoheadrightarrow^* C \overset{cmt}{\twoheadrightarrow} C' \twoheadrightarrow^* C_{rl} \overset{roll}{\rightsquigarrow} C''$$

with $C'' = C_{cmt}$. Now, consider the commit performed by C, supposing that it is triggered by P_c evolving to P_c' in doing that, we have that:

$$C \equiv \big(\nu s : C_0\big)(\langle P\rangle \blacktriangleright P_c \mid \langle Q\rangle \blacktriangleright Q_c) \ \text{ and } \ C' = \big(\nu s : C_0\big)(\langle P_c'\rangle \blacktriangleright P_c' \mid \langle Q_c\rangle \blacktriangleright Q_c)$$

Now, let us consider the case where $C' \twoheadrightarrow^* C_{rl}$ without any commit being present in the trace, hence:

$$C_{rl} \equiv \big(\nu s : C_0\big)(\langle P_c'\rangle \blacktriangleright P_{rl} \mid \langle Q_c\rangle \blacktriangleright Q_{rl})$$

$$\frac{P_1 \xrightarrow{k!\langle v\rangle} P_1' \quad \neg P_2 \Downarrow_{k?} \quad \neg P_2 \Downarrow_{roll}}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{com\_error}} \text{[E-Com}_1\text{]} \qquad \frac{P_1 \xrightarrow{k?(x)} P_1' \quad \neg P_2 \Downarrow_{k!} \quad \neg P_2 \Downarrow_{roll}}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{com\_error}} \text{[E-Com}_2\text{]}$$

$$\frac{P_1 \xrightarrow{k\triangleleft l} P_1' \quad \neg P_2 \Downarrow_{k\triangleright l} \quad \neg P_2 \Downarrow_{roll}}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{com\_error}} \text{[E-Lab}_1\text{]} \qquad \frac{P_1 \xrightarrow{k\triangleright l} P_1' \quad \neg P_2 \Downarrow_{k\triangleleft l} \quad \neg P_2 \Downarrow_{roll}}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{com\_error}} \text{[E-Lab}_2\text{]}$$

$$\frac{P_1 \xrightarrow{cmt} P_1' \qquad \tilde{Q_2} \neq P_2}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \langle P_1'\rangle \blacktriangleright P_1' \mid \langle \underline{P_2}\rangle \blacktriangleright P_2} \text{[E-Cmt}_1\text{]}$$

$$\frac{P_1 \xrightarrow{cmt} P_1' \qquad \tilde{Q_2} = P_2}{\langle \tilde{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \langle P_1'\rangle \blacktriangleright P_1' \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2} \text{[E-Cmt}_2\text{]}$$

$$\frac{P_1 \xrightarrow{roll} P_1'}{\langle Q_1\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \rightsquigarrow \langle Q_1\rangle \blacktriangleright Q_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright Q_2} \text{[E-Rll}_1\text{]}$$

$$\frac{P_1 \xrightarrow{roll} P_1'}{\langle \underline{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q_2}\rangle \blacktriangleright P_2 \twoheadrightarrow \texttt{roll\_error}} \text{[E-Rll}_2\text{]}$$

Figure 12: `cherry-pi` semantics: error reductions.

By hypothesis, from C_{rl} a rollback is possible. Regardless the rollback is triggered by P_{rl} or Q_{rl}, we have that $C_{rl} \overset{roll}{\rightsquigarrow} C'$, hence $C' \equiv C''$. Now, from C' we cannot reach C (i.e., $C' \not\twoheadrightarrow^+ C$), as C' is derived from C and the rollback can only bring the collaboration back to C'. Therefore, we have $C'' \not\twoheadrightarrow^+ C$, which violates the initial hypothesis.

Let us consider, instead, the case where $C' \twoheadrightarrow^* C_{rl}$ with at least a commit being present in the trace. By applying rule [B-Rll], from C_{rl} the rollback will lead to the last committed collaboration C'_{cmt} such that $C' \twoheadrightarrow^* \overset{cmt\,s}{\twoheadrightarrow} C'_{cmt} \twoheadrightarrow^* C_{rl}$. Hence, $C'_{cmt} \equiv C''$ and, like in the previous case, from C'_{cmt} we cannot reach C. Again we have $C'' \not\twoheadrightarrow^+ C$, which violates the initial hypothesis. Therefore, in each case the initial hypothesis is violated, and hence we conclude. □

6.3. **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 [CDSY17]) rather than on the usual static characterisation of errors (as, e.g., in [YV07]), 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 additional error reduction rules in Fig. 12, where [E-Cmt$_1$] and [E-Cmt$_2$] replace [F-Cmt], and [E-Rll$_1$] replaces [B-Rll]; moreover, $\tilde{}$ 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 *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 $k?$, $k!$,$k \triangleleft l$, $k \triangleright l$, 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 [E-Com$_1$]) or vice versa (rule [E-Com$_2$]), or one participant is willing to perform a selection but the corresponding branching is not available on the other side (rule [E-Lab$_1$]) or viceversa (rule [E-Lab$_2$]). 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 (rule [E-Rll$_2$]). To enable this error check, the rules for commit and rollback (rules [E-Cmt$_1$], [E-Cmt$_2$], and [E-Rll$_1$]) 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: $\mathbb{C} \ ::= \ [\cdot] \ \mid \ \mathbb{C} | C \ \mid \ \big(\nu s : C\big)\ \mathbb{C}$.

Definition 6.9 (Erroneous collaborations)**.** A collaboration C is *communication (resp. rollback) erroneous* if $C = \mathbb{C}[\texttt{com\_error}]$ (resp. $C = \mathbb{C}[\texttt{roll\_error}]$).

The key soundness results follow: a rollback safe collaboration never reduces to either a rollback erroneous collaboration (Theorem 6.10) or a communication erroneous collaboration (Theorem 6.11).

Theorem 6.10 (Rollback soundness)**.** *If C is a roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\texttt{roll\_error}]$.*

Proof (sketch). The proof proceeds by contradiction (the full proof is reported in Appendix A).

Theorem 6.11 (Communication soundness)**.** *If C is a roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\texttt{com\_error}]$.*

Proof (sketch). The proof proceeds by contradiction (the full proof is reported in Appendix A).

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 6.10 and 6.11, and from the fact that we consider binary sessions without delegation and subordinate sessions.

Theorem 6.12 (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 $C' \equiv \big(\nu s : C\big)(\langle\tilde{Q}_1\rangle \blacktriangleright \mathbf{0} \,|\, \langle\tilde{Q}_2\rangle \blacktriangleright \mathbf{0})$ for some $\tilde{Q}_1$ and $\tilde{Q}_2$.*

Proof. The proof proceeds by contradiction. Suppose that C is rollback safe and $C \rightarrowtail^* C'$ with $C' \not\rightarrowtail$ and $C' \not\equiv \big(\nu s : C\big)(\langle\tilde{Q}_1\rangle \blacktriangleright \mathbf{0} \ \mid \ \langle\tilde{Q}_2\rangle \blacktriangleright \mathbf{0})$ for any $\tilde{Q}_1$ and $\tilde{Q}_2$. The only

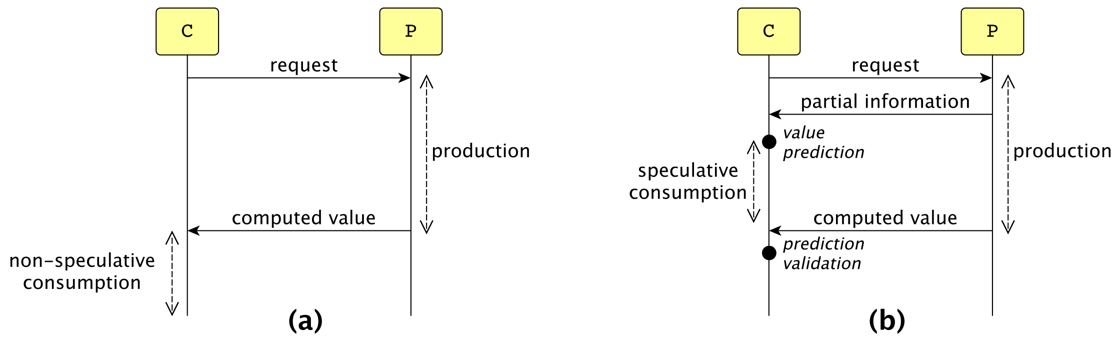

Figure 13: Producer-consumer scenario with non-speculative (a) and speculative (b) consumers.

situations that prevents C' from progressing are $C' = \mathbb{C}[\texttt{roll\_error}]$ or $C' = \mathbb{C}[\texttt{com\_error}]$. However, from Theorems 6.10 and 6.11, respectively, we have $C' \neq \mathbb{C}[\texttt{roll\_error}]$ and $C' \neq \mathbb{C}[\texttt{com\_error}]$, which is a contradiction. □

7. `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 [PRV10]. 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 [GLMT17] 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

$$\overline{start}(x).P_{\mathtt{C}} \mid start(y).P_{\mathtt{P}}$$

where the consumer and producer processes are:

$$P_{\mathsf{C}} = \mu X.\, x!\langle f_{req}()\rangle.\, x \rhd \{\, l_{spec} \,:\, x?(x_{partial} : \mathtt{str}).\, x?(x_{final} : \mathtt{str}).\ \mathsf{if}\ (f_{compare}(x_{partial}, x_{final}))\ \mathsf{then\ roll\ else\ commit}.\, X\,,\ l_{nonSpec} \,:\, x?(x_{computed} : \mathtt{str}).\, \mathsf{commit}.\, X\ \}$$

$$P_{\mathsf{P}} = \mu Y.\, y?(y_{req} : \mathtt{str}).\ \mathsf{if}\ (f_{eval}(y_{req}))\ \mathsf{then}\ y \lhd l_{spec}.\, y!\langle f_{partial}(y_{req})\rangle.\, y!\langle f_{final}(y_{req})\rangle.\, Y\ \mathsf{else}\ y \lhd l_{nonSpec}.\, y!\langle f_{compute}(y_{req})\rangle.\, Y$$

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.

To check whether the above collaboration is rollback safe, we have to check compliance between the session types of the two parties. The session types `Tc` and `Tp` associated by the type system to the consumer and the producer processes, respectively, and the corresponding initial type configuration are as follows (hereafter we use the MAUDE implementation of the `cherry-pi` type syntax):

```
eq Tc = mu X . ![str]. brn[brnEl('spec, ?[str]. ?[str]. (roll (+) (cmt. X)));
                           brnEl('nonSpec, ?[str]. cmt. X)] .

eq Tp = mu Y . ?[str]. ((sel['spec]. ![str]. ![str]. Y) (+) (sel['nonSpec].
                                                              ![str]. Y)) .

eq CPInitConfig = init(Tc,Tp) ckp(Tc) > Tc || ckp(Tp) > Tp .
```

As discussed in Sec. 5, the compliance check for the above type specification is performed by resorting to the MAUDE command `search` as follows:

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

The above command returns:

```
No solution.
```

meaning that the producer-consumer collaboration is rollback safe. To investigate more in detail the behaviour of this collaboration, we can generate the transition system associated to the type configuration `CPInitConfig` by using the following MAUDE commands:

```
search CPInitConfig =>* C:Configuration .
show search graph .
```

The first command searches for all states of the transition, reachable in none, one, or more steps, while the second command returns the current search graph generated by the previous search command. A graphical representation of the generated graph is reported in Fig. 14, where states represent the reachable type configurations (state 0 corresponds to `CPInitConfig`) and transitions are labelled by the applied rules (from Figure 11). Notably, in case a value is correctly consumed, since the commit action is performed at the end of the recursive step, the type configuration resulting from the commit coincides with the initial

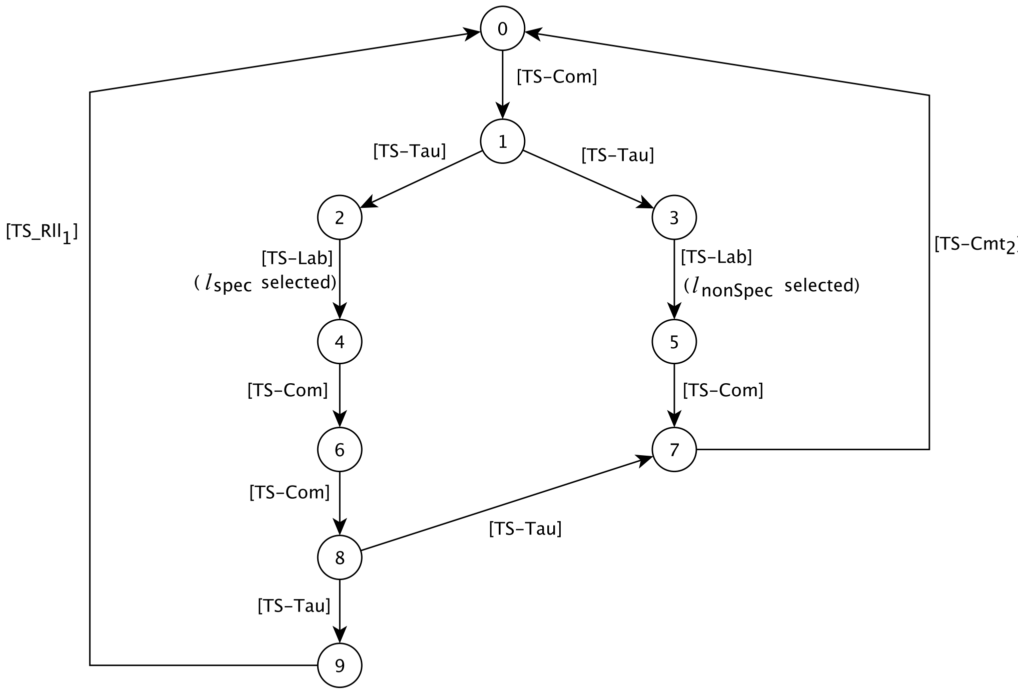

Figure 14: Transition system of the type configuration `CPInitConfig`.

one (see transition from state 7 to state 0 in the graph), as both consumer and producer are ready to repeat their interactions for a new value. In case of incorrect prediction (transition from state 8 to state 9), instead, the session execution is moved back to the last checkpoint (transition from state 9 to state 0), corresponding to the successfully consumption of the previous requested value that, at type level, correspond again to the initial type configuration. Indeed, as previously discussed, no commit occurs during the speculative consumption, hence no checkpoint has been created after the one recorded at the end of the previous recursive step. It is worth noticing that in case the prediction of the first produced value is wrong, and hence no commit action is performed by the consumer yet, according to the `cherry-pi` semantics the checkpoint corresponds to the beginning of the session.

Let us consider now a variant of the above scenario where the producer commits each time a value production is completed, which could apparently seem a reasonable behaviour from the producer side. The session type `Tp'` of such a producer process and the corresponding initial type configuration are as follows:

```
eq Tp' = mu Y . ?[str]. ((sel['spec]. ![str]. ![str]. cmt. Y)
                          (+) (sel['nonSpec]. ![str]. cmt. Y)) .

eq CPInitConfig' = init(Tc,Tp') ckp(Tc) > Tc || ckp(Tp') > Tp' .
```

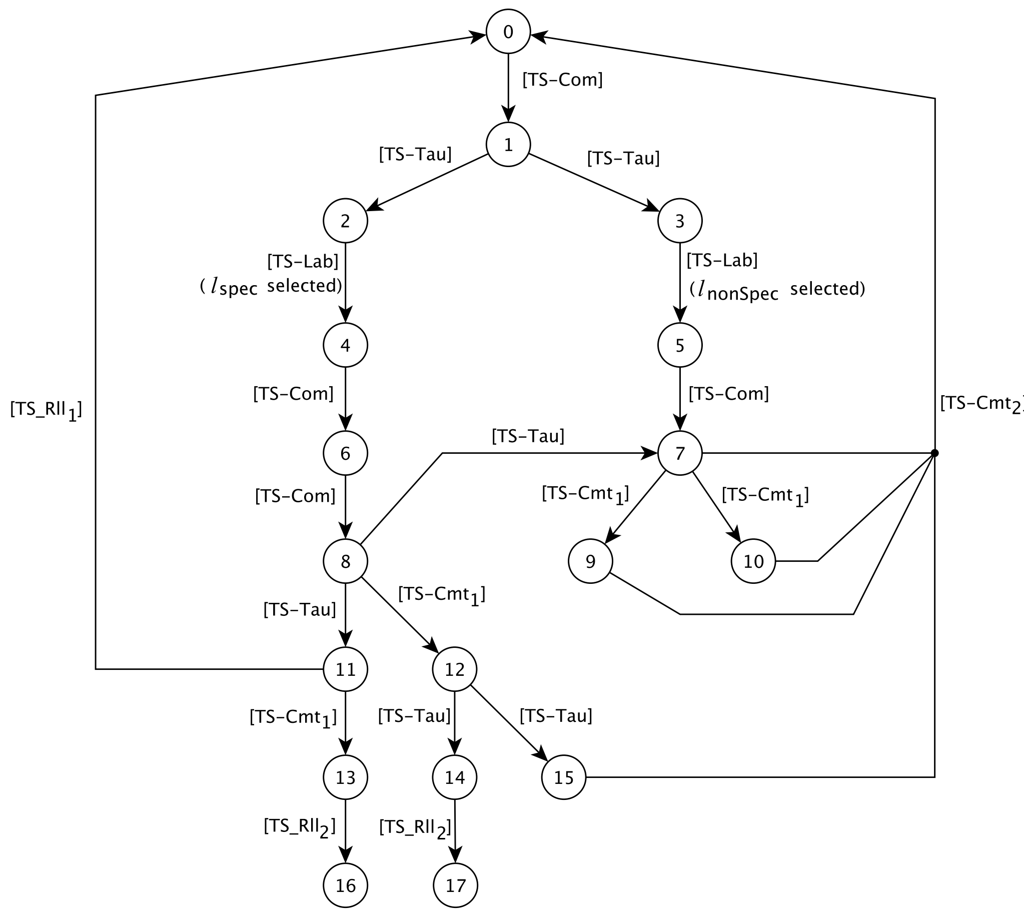

Figure 15: Transition system of the type configuration `CPInitConfig'`.

This time the collaboration is not rollback safe. Indeed, the compliance check returns two solutions:

```
Solution 1 (state 16)
CT1 --> ickp(roll)
T1 --> err
CT2 --> ckp(mu Y . ?[str]. ((sel['spec]. ![str]. ![str]. cmt. Y)
                            (+)sel['nonSpec]. ![str]. cmt. Y))
T2 --> err

Solution 2 (state 17)
CT1 --> ickp(roll(+)cmt. mu X . ![str]. brn[brnEl('nonSpec, ?[str]. cmt. X);
                                 brnEl('spec, ?[str]. ?[str]. (roll(+)cmt. X))])
T1 --> err
CT2 --> ckp(mu Y . ?[str]. ((sel['spec]. ![str]. ![str]. cmt. Y)
                            (+)sel['nonSpec]. ![str]. cmt. Y))
T2 --> err
```

corresponding to two erroneous configurations. The overall behaviour is graphically depicted by the transition system in Fig. 15, where state 0 corresponds to `CPInitConfig'` and states 16 and 17 are the two erroneous configurations above. While the commit action in the non-speculative case (transitions from state 7 to state 10, and from state 9 to state 0), does not affect the compliance between the two session participants, the other commit action (transitions from state 8 to state 12, and from state 11 to state 13), 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.

8. Related work

In the literature we can distinguish three ways of dealing with rollback: either using explicit rollbacks and implicit commits [LMSS11], or by using explicit commits and spontaneous aborts [DK05, dVKH10], or by just considering implicit rollbacks and commits [FV05, SKM17]. 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 [LMSS11]), while explicit commits limit the scope of potential future reverse executions (as in [DK05, dVKH10]). Differently from [DK05, dVKH10], 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* [BDd14, BDLd16, BLd17], *binary session types* [MP17], *multiparty session types* [MP21, CDG17, TY15, TY16], and *global graphs* [MT17, FMT18]. In [BDd14, BDLd16, BLd17] choices can be seen as *implicit* checkpoints and the system can go back to a previous choice and try another branch. In [BDd14] rollback is triggered non-deterministically (e.g., it may happen at any time during the execution), while in [BDLd16, BLd17] it is triggered by the system only when the forward computation is stuck (e.g., the client cannot further continue). 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 [MP17] reversibility is studied in the context of binary session types. Types information is used at run-time by monitors, for binary [MP17] and multiparty [MP21] 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 [CDG17] 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 [TY15] an *uncontrolled* reversible variant of session π-calculus is presented, while [TY16] studies different notions of reversibility for both binary and multiparty single sessions. In [MT17, FMT18, FMT20] 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

Work	Formalism	Commit	Rollback	Confined
[LMSS11]	Process Calculi	I	E	N
[DK05]	Process Calculi	E	I	Y
[dVKH10]	Process Calculi	E	I	N
[BDd14, BDLd16, BLd17]	Contracts	I	I	Y
[MP17, MP21]	Session Types	I	I	N
[CDG17]	Session Types	I	I	Y
[MT17, FMT18, FMT20]	Global Graphs	I	E	Y
[Vid23]	Process Calculi	E	E	N
[FV05]	Actor Model	E	E	Y
[SKM17]	Actor Model	I	I	Y
Our work	Actor Model & Session Types	E	E	N

Table 1: Approaches in the literature; commits and rollbacks can be either implicit (I) or explicit (E); reversibility can be confined (Y) or not (N).

conditions and constraints about loops have to be satisfied. Hence, in order to trigger a rollback a runtime condition should be satisfied.

The closest work in the literature to ours is [Vid23] where a checkpoint rollback facility is studied on top a reversible actor based language. Here, checkpoint and rollbacks primitives are explicit, as in our approach. Also, the proposed approach scales with dynamically created processes (e.g., spawning), while we do not deal with this issue. On the other hand, [Vid23] does not deal with ruling out undesired behaviours. Reversibility, in terms of transactional behaviours, has been investigated in the context of the actor model also in [FV05, SKM17]. While [FV05] introduces a facility to create a global checkpoint among multiple actors, [SKM17] studies the application of software transactional memory to the actor model. In [FV05] commits and rollbacks are explicit and confined inside the scope of a stabilisation of the actor state. On the other hand, in [SKM17] rollbacks and commits are implicit: if at the end the transaction can commit it does so, otherwise it is rolled back to the very beginning and re-tried again. Also, in this work reversibility is confined within the transactional scope. Let us note that [FV05] uses a very fine-grained mechanism to keep track of the causality graph among all the actors that have interacted with the one who wants to commit. We will further study this mechanism when adapting our theory to multi-party session types.

Table 1 summarises all the approaches in the literature. We detach from these works in several ways. Our checkpoint facility is explicit and checkpointing is not confined 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. Our compliance check, which is decidable, resembles those for contracts introduced in [BDd14, BDLd16, BLd17], which are defined for different rollback recovery approaches based on implicit checkpoints. Specifically, our compliance relation is similar to the ones defined in [BDLd16, BLd17] as they consist in requiring that, whenever no reduction is possible, all client requests and offers have been satisfied. Similarly, in our compliance notion, two participants are compliant if, whenever they reach a type configuration where no reduction is possible, they are in the successful (end) state. Our notion differs from the others as we do not distinguish client and server roles and, of course, we have different technicalities as our types (resp. type configurations) differ from retractable contracts (resp. client/server pairs). The compliance relation in [BDd14], instead, differs from the others (including ours) as it is coinductively defined. We have not considered this approach for our definition as it would make the theoretical framework much more complicated and, most of all, would be less suitable for a MAUDE implementation of the compliance checking.

Concerning the MAUDE implementation of the compliance check, we have followed the approach of the seminal work by Verdejo and Martí-Oliet [VM02], 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 [CFP$^+$18], Twitlang [MPST17], SCEL [BDVW14], and QFLan [VtBLL18].

9. Concluding remarks

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 6.6 and Theorem 6.8) and session soundness (Theorems 6.10 and 6.11). Our compliance check has been implemented in MAUDE.

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 [WS92]. 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 [EAWJ02]; we leave as future work a thoughtful comparison between these two mechanisms.

Another possible future work is to adapt our framework to work in asynchronous settings, like real applications. The first thing to do is to extend our framework with queues (either a global one or one queue per participant), in order to deal with asynchronous messages. We could adapt the work of [MP17, MP21] in which uncontrolled reversibility is added to asynchronous binary and multiparty session types. Adding controlled reversibility atop on them would require special messages for commit, abort and checkpoint. Such messages should be handled with priority, or at least one has to assume fairness, otherwise there will be no guarantee that a commit or a rollback will be performed. We could start from the asynchronous semantics given in [LMSS11]. Let us note that considering messages with priority is totally licit, as languages for large scale applications such as Erlang, Elixir and Akka allow for messages with priority. Starting from this settings and adding a fine-grained causality tracking mechanism, such as the one of [FV05], is our long-term goal, allowing us to have a complete theory for a real-world fault-tolerant applications.

Finally, we plan to investigate the practical application of our work to more realistic programming languages. Since our proposal has been devised for communication-centric systems, as a first testing ground we plan to transfer our ideas to a message-passing programming language based on the π-calculus, e.g. SePi[2] [FV13]. The challenge would be not only the extension of the language with our linguistic primitives to program reversible sessions, but also extending our results to a setting richer in terms of programming constructs and features. Then, we will consider programming languages that feature session-based programming, but are based on paradigms other than the π-calculus, e.g. *sessionj*[3] [HKP+10], an extension of Java with session-based constructs. Another direction would be the application of our approach to Scribble[4] [YHNN13], a framework to specify application-level protocols among communicating systems that supports bindings for several high-level languages.

Acknowledgment

The authors would like to acknowledge anonymous reviewers for their useful criticisms and suggestions that have greatly contributed to improve the paper.

References

[ACG+20] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Lukasz Mikulski, Rajagopal Nagarajan, Iain C. C. Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski, and Germán Vidal. Foundations of reversible computation. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz, and Carla Ferreira, editors, *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *Lecture Notes in Computer Science*, pages 1–40. Springer, 2020. doi:10.1007/978-3-030-47361-7_1.

[BDd14] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Compliance for reversible client/server interactions. In Marco Carbone, editor, *BEAT*, volume 162 of *EPTCS*, pages 35–42, 2014. doi:10.4204/EPTCS.162.5.

[BDLd16] Franco Barbanera, Mariangiola Dezani-Ciancaglini, Ivan Lanese, and Ugo de'Liguoro. Retractable contracts. In Simon Gay and Jade Alglave, editors, *PLACES 2015*, volume 203 of *EPTCS*, pages 61–72, 2016. doi:10.4204/EPTCS.203.5.

[BDVW14] Lenz Belzner, Rocco De Nicola, Andrea Vandin, and Martin Wirsing. Reasoning (on) service component ensembles in rewriting logic. In *Specification, Algebra, and Software*, volume 8373 of *Lecture Notes in Computer Science*, pages 188–211. Springer, 2014. doi:10.1007/978-3-319-16310-9.

[BLd17] Franco Barbanera, Ivan Lanese, and Ugo de'Liguoro. Retractable and speculative contracts. In Jean-Marie Jacquet and Mieke Massink, editors, *COORDINATION*, volume 10319 of *Lecture Notes in Computer Science*, pages 119–137. Springer, 2017. doi:10.1007/978-3-319-59746-1_7.

[BLd18] Franco Barbanera, Ivan Lanese, and Ugo de'Liguoro. A theory of retractable and speculative contracts. *Sci. Comput. Program.*, 167:25–50, 2018. URL: https://doi.org/10.1016/j.scico.2018.06.005, doi:10.1016/J.SCICO.2018.06.005.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. doi:10.1007/978-3-540-71999-1.

[CDG17] Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Concurrent reversible sessions. In Roland Meyer and Uwe Nestmann, editors, *CONCUR*, volume 85 of *LIPIcs*, pages 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. URL: https://doi.org/10.4230/LIPIcs.CONCUR.2017.30, doi:10.4230/LIPICS.CONCUR.2017.30.

[2] https://rss.rd.ciencias.ulisboa.pt/tools/sepi/

[3] https://code.google.com/archive/p/sessionj/

[4] https://github.com/scribble

[CDSY17] Tzu-Chun Chen, Mariangiola Dezani-Ciancaglini, Alceste Scalas, and Nobuko Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017. doi:10.23638/LMCS-13(2:12)2017.

[CFP+18] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, and Francesco Tiezzi. A formal approach to modeling and verification of business process collaborations. *Sci. Comput. Program.*, 166:35–70, 2018. URL: https://doi.org/10.1016/j.scico.2018.05.008, doi:10.1016/J.SCICO.2018.05.008.

[CKV13] Ioana Cristescu, Jean Krivine, and Daniele Varacca. A Compositional Semantics for the Reversible p-Calculus. In *LICS*, pages 388–397. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.45.

[DK04] Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.

[DK05] Vincent Danos and Jean Krivine. Transactions in rccs. In Martín Abadi and Luca de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005. doi:10.1007/11539452_31.

[dVKH10] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. Communicating transactions - (extended abstract). In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-15375-4_39.

[EAWJ02] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002. doi:10.1145/568522.568525.

[Eng12] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D)*, pages 1–6, Sept 2012.

[FMT18] Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Reversible choreographies via monitoring in erlang. In Silvia Bonomi and Etienne Rivière, editors, *Distributed Applications and Interoperable Systems - 18th IFIP WG 6.1 International Conference, DAIS 2018*, volume 10853 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2018. doi:10.1007/978-3-319-93767-0_6.

[FMT20] Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto. Towards choreographic-based monitoring. In *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *Lecture Notes in Computer Science*, pages 128–150. Springer, 2020. doi:10.1007/978-3-030-47361-7_6.

[FV05] John Field and Carlos A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005*, pages 195–208. ACM, 2005. doi:10.1145/1040305.1040322.

[FV13] Juliana Franco and Vasco Thudichum Vasconcelos. A concurrent programming language with refined session types. In Steve Counsell and Manuel Núñez, editors, *SEFM 2013*, volume 8368 of *Lecture Notes in Computer Science*, pages 15–28. Springer, 2013. doi:10.1007/978-3-319-05032-4_2.

[GLM14] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014. doi:10.1007/978-3-642-54804-8_26.

[GLMT17] Elena Giachino, Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebr. Meth. Program.*, 88:99–120, 2017. URL: https://doi.org/10.1016/j.jlamp.2016.09.003, doi:10.1016/J.JLAMP.2016.09.003.

[HKP+10] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In Theo D'Hondt, editor, *ECOOP 2010*, volume 6183 of *Lecture Notes in Computer Science*, pages 329–353. Springer, 2010. doi:10.1007/978-3-642-14107-2_16.

[HLV+16] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

[HVK98] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. URL: `https://doi.org/10.1007/BFb0053558`, `doi:10.1007/BFB0053558`.

[KU18] Stefan Kuhn and Irek Ulidowski. Local reversibility in a calculus of covalent bonding. *Sci. Comput. Program.*, 151:18–47, 2018. URL: `https://doi.org/10.1016/j.scico.2017.09.008`, `doi:10.1016/J.SCICO.2017.09.008`.

[LLM+13] Ivan Lanese, Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Concurrent flexible reversibility. In Matthias Felleisen and Philippa Gardner, editors, *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 370–390. Springer, 2013. `doi:10.1007/978-3-642-37036-6\_21`.

[LMS10] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversing higher-order pi. In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2010. `doi:10.1007/978-3-642-15375-4\_33`.

[LMSS11] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Controlling Reversibility in Higher-Order Pi. In Joost-Pieter Katoen and Barbara König, editors, *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. `doi:10.1007/978-3-642-23217-6\_20`.

[LSU22] Ivan Lanese, Ulrik Pagh Schultz, and Irek Ulidowski. Reversible computing in debugging of erlang programs. *IT Prof.*, 24(1):74–80, 2022. `doi:10.1109/MITP.2021.3117920`.

[Mes92] José Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992. `doi:10.1016/0304-3975(92)90182-F`.

[Mes12] José Meseguer. Twenty years of rewriting logic. *J. Log. Algebraic Methods Program.*, 81(7-8):721–781, 2012. URL: `https://doi.org/10.1016/j.jlap.2012.06.003`, `doi:10.1016/J.JLAP.2012.06.003`.

[Mil89] Robin Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[MP17] Claudio Antares Mezzina and Jorge A. Pérez. Reversibility in session-based concurrency: A fresh look. *J. Log. Algebraic Methods Program.*, 90:2–30, 2017. URL: `https://doi.org/10.1016/j.jlamp.2017.03.003`, `doi:10.1016/J.JLAMP.2017.03.003`.

[MP21] Claudio Antares Mezzina and Jorge A. Pérez. Causal consistency for reversible multiparty protocols. *Log. Methods Comput. Sci.*, 17(4), 2021. URL: `https://doi.org/10.46298/lmcs-17(4:1)2021`, `doi:10.46298/LMCS-17(4:1)2021`.

[MPST17] Alessandro Maggi, Marinella Petrocchi, Angelo Spognardi, and Francesco Tiezzi. A language-based approach to modelling and analysis of twitter interactions. *J. Log. Algebr. Meth. Program.*, 87:67–91, 2017. URL: `https://doi.org/10.1016/j.jlamp.2016.11.003`, `doi:10.1016/J.JLAMP.2016.11.003`.

[MSG+20] Claudio Antares Mezzina, Rudolf Schlatte, Robert Glück, Tue Haulund, James Hoey, Martin Holm Cservenka, Ivan Lanese, Torben Æ. Mogensen, Harun Siljak, Ulrik Pagh Schultz, and Irek Ulidowski. Software and reversible systems: A survey of recent activities. In Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz, and Carla Ferreira, editors, *Reversible Computation: Extending Horizons of Computing - Selected Results of the COST Action IC1405*, volume 12070 of *Lecture Notes in Computer Science*, pages 41–59. Springer, 2020. `doi:10.1007/978-3-030-47361-7\_2`.

[MT17] Claudio Antares Mezzina and Emilio Tuosto. Choreographies for automatic recovery. *CoRR*, abs/1705.09525, 2017. URL: `http://arxiv.org/abs/1705.09525`, `arXiv:1705.09525`.

[MTY23] Claudio Antares Mezzina, Francesco Tiezzi, and Nobuko Yoshida. Rollback recovery in session-based programming. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023*, volume 13908 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2023. `doi:10.1007/978-3-031-35361-1\_11`.

[PP13] Kalyan S. Perumalla and Vladimir A. Protopopescu. Reversible simulations of elastic collisions. *ACM Trans. Model. Comput. Simul.*, 23(2):12:1–12:25, 2013. `doi:10.1145/2457459.2457461`.

[PRV10] Prakash Prabhu, Ganesan Ramalingam, and Kapil Vaswani. Safe programmable speculative parallelism. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 50–61. ACM, 2010. `doi:10.1145/1806596.1806603`.

[SKM17] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. Transactional actors: communication in transactions. In Ali Jannesari, Pablo de Oliveira Castro, Yukinori Sato, and Tim Mattson, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems, SEPS@SPLASH 2017*, pages 31–41. ACM, 2017. doi:10.1145/3141865.3141866.

[TY15] Francesco Tiezzi and Nobuko Yoshida. Reversible session-based pi-calculus. *J. Log. Algebr. Meth. Program.*, 84(5):684–707, 2015. URL: https://doi.org/10.1016/j.jlamp.2015.03.004, doi:10.1016/J.JLAMP.2015.03.004.

[TY16] Francesco Tiezzi and Nobuko Yoshida. Reversing single sessions. In Simon J. Devitt and Ivan Lanese, editors, *RC*, volume 9720 of *Lecture Notes in Computer Science*, pages 52–69. Springer, 2016. doi:10.1007/978-3-319-40578-0_4.

[Vas21] Martin Vassor. Reversibility and predictions. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 163–181. Springer, 2021. doi:10.1007/978-3-030-79837-6_10.

[Vid23] Germán Vidal. From reversible computation to checkpoint-based rollback recovery for message-passing concurrent programs. In Javier Cámara and Sung-Shik Jongmans, editors, *Formal Aspects of Component Software - 19th International Conference, FACS 2023, Virtual Event, October 19-20, 2023, Revised Selected Papers*, volume 10853 of *Lecture Notes in Computer Science*, pages 103–123. Springer, 2023. doi:10.1007/978-3-031-52183-6_6.

[VM02] Alberto Verdejo and Narciso Martí-Oliet. Implementing CCS in maude 2. In *Fourth International Workshop on Rewriting logic and Its Applications, WRLA2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 282–300. Elsevier, 2002. doi:10.1016/S1571-0661(05)82540-X.

[VS18] Martin Vassor and Jean-Bernard Stefani. Checkpoint/rollback vs causally-consistent reversibility. In *Reversible Computation - 10th International Conference, RC 2018,*, volume 11106 of *Lecture Notes in Computer Science*, pages 286–303. Springer, 2018. doi:10.1007/978-3-319-99498-7_20.

[VtBLL18] Andrea Vandin, Maurice H. ter Beek, Axel Legay, and Alberto Lluch-Lafuente. Qflan: A tool for the quantitative analysis of highly reconfigurable systems. In *FM*, volume 10951 of *Lecture Notes in Computer Science*, pages 329–337. Springer, 2018. doi:10.1007/978-3-319-95582-7.

[WS92] Gerhard Weikum and Hans-J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan Kaufmann, 1992.

[YHNN13] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *TGC 2013*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

[YV07] Nobuko Yoshida and Vasco Thudichum Vasconcelos. 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. URL: https://doi.org/10.1016/j.entcs.2007.02.056, doi:10.1016/J.ENTCS.2007.02.056.

Appendix A. Proofs

The auxiliary lemmata required to prove the soundness results 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 $\tilde{P} = \underline{P}$ implies $\tilde{T} = \underline{T}$;
- let ℓ a process label, its *dual label* $\bar{\ell}$ is defined as follows: $\overline{k!\langle v\rangle} = k?(x)$ for some x, $\overline{k?(x)} = k!\langle v\rangle$ for some v, $\overline{k \lhd l} = k \rhd l$, $\overline{k \rhd l} = k \lhd l$; this notion of duality straightforwardly extends to type labels;
- the function $tl_\Gamma(\cdot)$, mapping process labels to type labels under sorting Γ, is defined as follows: $tl_\Gamma(k!\langle v\rangle) = ![S]$ with $\Gamma \ \vdash \ v \ \blacktriangleright \ S$, $tl_\Gamma(k?(x)) = ?[S]$ with $\Gamma \ \vdash \ x \ \blacktriangleright \ S$, $tl_\Gamma(k \lhd l) = \lhd l$, $tl_\Gamma(k \rhd l) = \rhd l$, $tl_\Gamma(cmt) = \texttt{cmt}$, $tl_\Gamma(roll) = \texttt{roll}$, $tl_\Gamma(abt) = \texttt{abt}$, and $tl_\Gamma(\tau) = \tau$.

The following lemma states that each reduction of a reachable collaboration corresponds to a reduction of its configuration types.

Lemma A.1. *Let* $C = \big(\nu s : C'\big)(\langle\tilde{P_1}\rangle \blacktriangleright P_2 \ \mid \ \langle\tilde{Q_1}\rangle \blacktriangleright Q_2)$ *be a reachable collaboration, with* $C' = (\bar{a}(x).P \mid a(y).Q)$, $(\varnothing;\varnothing \ \vdash \ P \ \blacktriangleright \ x : T)$, $(\varnothing;\varnothing \ \vdash \ Q \ \blacktriangleright \ y : T')$, $T \dashv\vdash T'$, $(\Theta_1;\Gamma_1 \ \vdash \ P_1[x/\bar{s}] \ \blacktriangleright \ x : T_1)$, $(\Theta_2;\Gamma_2 \ \vdash \ P_2[x/\bar{s}] \ \blacktriangleright \ x : T_2)$, $(\Theta'_1;\Gamma'_1 \ \vdash \ Q_1[y/s] \ \blacktriangleright \ y : U_1)$, *and* $(\Theta'_2;\Gamma'_2 \ \vdash \ Q_2[y/s] \ \blacktriangleright \ y : U_2)$. *If* $C \rightarrowtail \big(\nu s : C'\big)(\langle\tilde{P'_1}\rangle \blacktriangleright P'_2 \ \mid \ \langle\tilde{Q'_1}\rangle \blacktriangleright Q'_2)$ *then there exist* T'_1, T'_2, U'_1, U'_2 *such that* $(T,T') : \langle\tilde{T_1}\rangle \blacktriangleright T_2 \parallel \langle\tilde{U_1}\rangle \blacktriangleright U_2 \longmapsto (T,T') : \langle\tilde{T'_1}\rangle \blacktriangleright T'_2 \parallel \langle\tilde{U'_1}\rangle \blacktriangleright U'_2$ *with* $\tilde{P'_1}$ *(resp.* $\tilde{Q'_1}$*) in checkpoint accordance with* $\tilde{T'_1}$ *(resp.* $\tilde{U'_1}$*)*, $(\hat{\Theta}_1;\hat{\Gamma}_1 \ \vdash \ P'_1[x/\bar{s}] \ \blacktriangleright \ x : T'_1)$, $(\hat{\Theta}_2;\hat{\Gamma}_2 \ \vdash \ P'_2[x/\bar{s}] \blacktriangleright x : T'_2)$, $(\hat{\Theta}'_1;\hat{\Gamma}'_1 \ \vdash \ Q'_1[y/s] \blacktriangleright y : U'_1)$, *and* $(\hat{\Theta}'_2;\hat{\Gamma}'_2 \ \vdash \ Q'_2[y/s] \blacktriangleright y : U'_2)$.

Proof. We have two cases depending whether the reduction $\rightarrowtail$ has forward or backward direction.

($\rightarrowtail = \twoheadrightarrow$)**:** From rule [F-Res], we have $\langle\tilde{P_1}\rangle \blacktriangleright P_2 \ \mid \ \langle\tilde{Q_1}\rangle \blacktriangleright Q_2 \twoheadrightarrow \langle\tilde{P'_1}\rangle \blacktriangleright P'_2 \ \mid \ \langle\tilde{Q'_1}\rangle \blacktriangleright Q'_2$. 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 $P_2 \xrightarrow{\bar{s}!\langle v\rangle} P'_2$, $Q_2 \xrightarrow{s?(z)} Q''_2$, with $Q'_2 = Q''_2[v/z]$, $\tilde{P'_1} = \tilde{P_1}$ and $\tilde{Q'_1} = \tilde{Q_1}$. Thus, $P_2[x/\bar{s}] = x!\langle e\rangle.P'_2[x/\bar{s}]$ for some e such that $e \downarrow v$, and $Q_2[y/s] = y?(z : S').Q'_2[y/s]$. By rule [T-Snd], we have that $T_2 = ![S].T'_2$, with $\Gamma_2 \ \vdash \ e \ \blacktriangleright \ S$, (hence $\Gamma_2 \ \vdash \ v \ \blacktriangleright \ S$), and $\Theta_2;\Gamma_2 \ \vdash \ P'_2[x/\bar{s}] \ \blacktriangleright \ x : T'_2$ (hence $\hat{\Theta}_2 = \Theta_2$ and $\hat{\Gamma}_2 = \Gamma_2$). Similarly, by rule [T-Rcv], we have that $U_2 = ?[S'].U'_2$ and $\Theta'_2;\Gamma'_2 \cdot z : S' \ \vdash \ Q''_2[y/s] \ \blacktriangleright \ y : U'_2$ (hence $\hat{\Theta}'_2 = \Theta'_2$ and $\hat{\Gamma}'_2 = \Gamma'_2 \cdot z : S'$). By rules [TS-Snd] and [TS-Rcv], we get $T_2 \xrightarrow{![S]} T'_2$ and $U_2 \xrightarrow{?[S']} U_2$. Now, reasoning by contradiction, let us suppose that $S \neq S'$. Thus, the term $(T,T') : \langle\tilde{T_1}\rangle \blacktriangleright T_2 \parallel \langle\tilde{U_1}\rangle \blacktriangleright U_2 \not\longmapsto$, since no rule in Fig. 11 can be applied. However, since C is a reachable collaboration, this type configuration is originated from $(T,T') : \langle T\rangle \blacktriangleright T \parallel \langle T'\rangle \blacktriangleright T'$. By Def. 4.1, $T \dashv\vdash T'$ implies $T_2 = U_2 = \texttt{end}$, which is a contradiction. Therefore, it holds that $S = S'$. Hence, by applying rule [TS-Com] we can conclude.
- [F-Lab], [E-Cmt$_1$] and [E-Cmt$_2$]. Similar to the previous case.
- [F-Par]. In this case we have that $\langle\tilde{P_1}\rangle \blacktriangleright P_2 \twoheadrightarrow \langle\tilde{P'_1}\rangle \blacktriangleright P'_2$. Since this transition involves only one log term, it can be inferred only by applying rule [F-If], from which we have $P_2 \xrightarrow{\tau} P'_2$ and $\tilde{P'_1} = \tilde{P_1}$. By rule [P-IfT] (the case of rule [P-IfF] is similar), we have $P_2[x/\bar{s}] = \texttt{if}\ e\ \texttt{then}\ P'_2[x/\bar{s}]\ \texttt{else}\ R$ with $e \downarrow \texttt{true}$. By rule [T-If]

we get $T_2 = T_2' \oplus V$ and $\Theta_2; \Gamma_2 \vdash P_2'[x/\bar{s}] \blacktriangleright x : T_2'$. By rule [TS-IF$_1$], $T_2 \xrightarrow{\tau} T_2'$. By applying rule [TS-TAU] we can conclude.

($\rightarrowtail = \rightsquigarrow$): The reduction can be inferred by applying rule [B-RES] or rule [B-ABT]. Let us consider the former case, the latter is similar. From rule [B-RES], we have $\langle \tilde{P_1} \rangle \blacktriangleright P_2 \mid \langle \tilde{Q_1} \rangle \blacktriangleright Q_2 \rightsquigarrow \langle \tilde{P_1'} \rangle \blacktriangleright P_2' \mid \langle \tilde{Q_1'} \rangle \blacktriangleright Q_2'$. We prove the result by case analysis on the last rule applied in the inference of the above reduction.

- [B-RLL]. In this case $P_2 \xrightarrow{roll} P_2'$, $\tilde{P}_1' = \tilde{P}_1$, $\tilde{Q}_1' = \tilde{Q}_1$ and $Q_2' = Q_1$. By rule [P-RLL], we have $P_2[x/\bar{s}] = \mathsf{roll}$ and $P_2'[x/\bar{s}] = \mathbf{0}$. By rule [T-RLL] we get $T_2 = \texttt{roll}$. By hypothesis $T \dashv\vdash T'$, which implies that rule [TS-RLL$_2$] is not applicable, because by Def. 4.1 an erroneous type configuration cannot be reached. Hence, by rule [TS-RLL$_1$], $T_2 \xrightarrow{roll} T_2'$, with $T_2' = \mathsf{end}$. By rule [T-INACT], we have $\Theta_2; \Gamma_2 \vdash P_2'[x/\bar{s}] \blacktriangleright x : T_2'$. Finally, by applying rule [TS-RLL$_1$] we can conclude.
- [B-PAR]. Similarly to the forward case. □

The following lemma relates collaboration reductions to type reductions when a `roll_error` is produced.

Lemma A.2. *Let $C = (\bar{a}(x).P \mid a(y).Q)$ such that $C \blacktriangleright \{\bar{a} : T_1, a : T_2\}$ and $T_1 \dashv\vdash T_2$. If $C \rightarrowtail^* \big(\nu s : C\big)(\langle \underline{P_1} \rangle \blacktriangleright P_2 \mid \langle \tilde{Q_1} \rangle \blacktriangleright Q_2)$ and $P_2 \xrightarrow{roll} P_2'$, then there exist U_1, U_2, U_1', U_2', U_1'' such that $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle \underline{U_1} \rangle \blacktriangleright U_1' \parallel \langle \tilde{U_2} \rangle \blacktriangleright U_2'$ and $U_1' \xrightarrow{roll} U_1''$.*

Proof. From $C \blacktriangleright \{\bar{a} : T_1, a : T_2\}$, by applying [T-PAR], [T-ACC] and [T-REQ], we have that $\varnothing; \varnothing \vdash P \blacktriangleright x : T_1$ and $\varnothing; \varnothing \vdash Q \blacktriangleright x : T_2$. By applying rule [F-CON] to the collaboration C, we obtain $C \twoheadrightarrow \big(\nu s : C\big)(\langle P[\bar{s}/x] \rangle \blacktriangleright P[\bar{s}/x] \mid \langle Q[s/y] \rangle \blacktriangleright Q[s/y]) = C'$. Now, by repeatedly applying Lemma A.1, from $C' \rightarrowtail^* \big(\nu s : C\big)(\langle \underline{P_1} \rangle \blacktriangleright P_2 \mid \langle \tilde{Q_1} \rangle \blacktriangleright Q_2)$ we get $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle \underline{U_1} \rangle \blacktriangleright U_1' \parallel \langle \tilde{U_2} \rangle \blacktriangleright U_2'$ for some U_1, U_2, U_1', U_2', with $\Theta_2; \Gamma_2 \vdash P_2[x/\bar{s}] \blacktriangleright x : U_1'$. Now, let us consider the transition $P_2 \xrightarrow{roll} P_2'$. This can be derived only by the application of rules [P-RLL]. Thus, $P_2 = \mathsf{roll}$, from which we have $P_2[x/\bar{s}] = \mathsf{roll}$. From $\Theta_2; \Gamma_2 \vdash \mathsf{roll} \blacktriangleright x : U_1'$, by rule [T-RLL], we get $U_1' = \texttt{roll}$. Therefore, by rule [TS-RLL$_2$], we can conclude $U_1' \xrightarrow{roll} U_1''$ with $U_1'' = \mathsf{end}$. □

The following lemma relates collaboration reductions to type reductions when a `com_error` is produced.

Lemma A.3. *Let $C = (\bar{a}(x).P \mid a(y).Q)$ such that $C \blacktriangleright \{\bar{a} : T_1, a : T_2\}$ and $T_1 \dashv\vdash T_2$. If $C \rightarrowtail^* \big(\nu s : C\big)(\langle \tilde{P_1} \rangle \blacktriangleright P_2 \mid \langle \tilde{Q_1} \rangle \blacktriangleright Q_2)$, $P_2 \xrightarrow{\ell} P_2'$ and $\neg Q_2 \Downarrow_{\bar{\ell}}$ with ℓ of the form $k!\langle v \rangle$, $k?(x)$, $k \lhd l$ or $k \rhd l$, then there exist U_1, U_2, U_1', U_2', U_1'' such that $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle \tilde{U_1} \rangle \blacktriangleright U_1' \parallel \langle \tilde{U_2} \rangle \blacktriangleright U_2'$ with $\tilde{P}_1$ (resp. $\tilde{Q}_1$) in checkpoint accordance with $\tilde{U}_1$ (resp. $\tilde{U}_2$), $U_1' \xrightarrow{tl_\Gamma(\ell)} U_1''$, with Γ sorting for typing P_2', and for all U_2'' such that $U_2' \xrightarrow{\tau}^* U_2''$ we have $U_2'' \overset{\overline{tl_\Gamma(\ell)}}{\not\longrightarrow}$.*

Proof. From $C \blacktriangleright \{\bar{a} : T_1, a : T_2\}$, by applying [T-PAR], [T-ACC] and [T-REQ], we have that $\varnothing; \varnothing \vdash P \blacktriangleright x : T_1$ and $\varnothing; \varnothing \vdash Q \blacktriangleright x : T_2$. By applying rule [F-CON] to the collaboration C, we obtain $C \twoheadrightarrow \big(\nu s : C\big)(\langle P[\bar{s}/x] \rangle \blacktriangleright P[\bar{s}/x] \mid \langle Q[s/y] \rangle \blacktriangleright Q[s/y]) = C'$. Now, by repeatedly applying Lemma A.1, from $C' \rightarrowtail^* \big(\nu s : C\big)(\langle \tilde{P_1} \rangle \blacktriangleright P_2 \mid \langle \tilde{Q_1} \rangle \blacktriangleright Q_2)$ we get $(T_1, T_2) : \langle T_1 \rangle \blacktriangleright T_1 \parallel \langle T_2 \rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle \tilde{U_1} \rangle \blacktriangleright U_1' \parallel \langle \tilde{U_2} \rangle \blacktriangleright U_2'$ for some U_1, U_2, U_1',

U_2', with $\Theta_2; \Gamma_2 \vdash P_2[x/\bar{s}] \blacktriangleright x : U_1'$ and $\Theta_2'; \Gamma_2' \vdash Q_2[y/s] \blacktriangleright y : U_2'$. Now, let us reason by case analysis on the rule for deriving the transition $P_2 \xrightarrow{\ell} P_2'$.

Rule [P-Snd]**:** Thus, $P_2 = \bar{s}!\langle e\rangle.P_2'$ and $\ell = \bar{s}!\langle v\rangle$ with $e \downarrow v$. From $\Theta_2; \Gamma_2 \vdash P_2[x/\bar{s}] \blacktriangleright x : U_1'$, by rule [T-Snd], we get $U_1' = ![S].U_1''$ with $\Gamma \vdash e \blacktriangleright S$. Therefore, by rule [TS-Snd], we get $U_1' \xrightarrow{![S]} U_1''$.

Rule [P-Rcv]**:** Thus, $P_2 = \bar{s}?(y : S).P_2'$ and $\ell = \bar{s}?(y)$. From $\Theta_2; \Gamma_2 \vdash P_2[x/\bar{s}] \blacktriangleright x : U_1'$, by rule [T-Rcv], we get $U_1' = ?[S].U_1''$. Therefore, by rule [TS-Rcv], we get $U_1' \xrightarrow{?[S]} U_1''$.

Rules [P-Sel] **and** [P-Brn]**:** Similar to the previous cases.

Finally, from $\neg Q_2 \Downarrow_{\bar{\ell}}$, following a similarly reasoning, we can conclude $U_2' \xrightarrow{\tau}{}^{*} \not\xrightarrow{\overline{tl_\Gamma(\ell)}}$. ∎

The soundness results are as follows.

Theorem 6.10. If C is a roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$.

Proof. The proof proceeds by contradiction. Suppose that there exists an initial collaboration C that is rollback safe and such that $C \rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$. The erroneous collaboration $\mathtt{roll\_error}$ can be only produced by applying rule [E-Rll$_2$]. Thus, to infer at least one reduction of the sequence $C \rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$, rule [E-Rll$_2$] must be used. From this, we have that there exists a runtime collaboration $C' \equiv \mathbb{C}'[C'']$, with $C'' = (\langle \underline{Q_1}\rangle \blacktriangleright P_1 \mid \langle \tilde{Q}_2\rangle \blacktriangleright P_2)$, such that $C \rightarrowtail^* C'$, $P_1 \xrightarrow{roll} P_1'$, and $C' \twoheadrightarrow \mathbb{C}'[\mathtt{roll\_error}] \rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$. By rules [F-Con], [F-Res] and [B-Res], and the fact that the scope of operator $(\nu s : \_)$ 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 $(\nu s : C_1)$, i.e. $\mathbb{C}' = C_2 \mid (\nu s : C_1)[\cdot]$, with $C_1 = \bar{a}(x).P \mid a(y).Q$ for some a, x, y, P and Q. In its own turn, the term $(\nu s : C_1)C''$ can only be generated by applying rule [F-Con] from C_1, which must be a subterm of C, i.e. $C \equiv C_1 \mid C_2'$ for some C_2'. Since the scope of $(\nu s : \_)$ 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 C_1 of C, by exploiting rules [F-Par] and [B-Par] we can set apart the reductions in $C \rightarrowtail^* \mathbb{C}[\mathtt{roll\_error}]$ involving C_1 and its derivatives, thus obtaining $C_1 \rightarrowtail^* (\nu s : C_1)C'' \twoheadrightarrow (\nu s : C_1)\mathtt{roll\_error}$.

Now, since C is rollback safe, by Def. 4.3 we have that $C \blacktriangleright A$ and for all pairs $\bar{b} : V_1$ and $b : V_2$ in A we have $V_1 \dashv\vdash V_2$. Since $C \equiv C_1 \mid C_2'$, by rule [T-Par] we obtain $C_1 \blacktriangleright A_1$ with $A_1 \subseteq A$. By rules [T-Req] and [T-Acc], we have $A_1 = \{\bar{a} : T_1, a : T_2\}$. Since A_1 is a subset of A, we have that $T_1 \dashv\vdash T_2$.

By Lemma A.2, we have that there exist $U_1, U_2, U_1', U_2', U_1''$ such that $(T_1, T_2) : \langle T_1\rangle \blacktriangleright T_1 \parallel \langle T_2\rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle \underline{U_1}\rangle \blacktriangleright U_1' \parallel \langle \tilde{U}_2\rangle \blacktriangleright U_2' = t$ and $U_1' \xrightarrow{roll} U_1''$. Since U_1' can only perform *roll*, the only rules that can be applied are [TS-Rll$_1$] and [TS-Rll$_2$]. However, rule [TS-Rll$_1$] cannot be applied due to the imposed checkpoint $\underline{U_1}$. Therefore, the only rule that can be applied is [TS-Rll$_2$], leading to the configuration $t' = (T_1, T_2) : \langle \underline{U_1}\rangle \blacktriangleright U_1'' \parallel \langle \tilde{U}_2\rangle \blacktriangleright U_2''$ with $U_1'' = U_2'' = \mathtt{err}$. Now, no rule in Fig. 11 allows the term t' to evolve, i.e. $t' \not\longmapsto$. Since $T_1 \dashv\vdash T_2$, by Def. 4.1 it must hold that $U_1'' = U_2'' = \mathtt{end}$. However, $U_1'' = \mathtt{err} \neq \mathtt{end}$ and $U_2'' = \mathtt{err} \neq \mathtt{end}$, which is a contradiction. ∎

Theorem 6.11. If C is a roll-safe collaboration, then we have that $C \not\rightarrowtail^* \mathbb{C}[\mathtt{com\_error}]$.

Proof. The proof proceeds by contradiction. Suppose that there exists an initial collaboration C that is rollback safe and such that $C \rightarrowtail^* \mathbb{C}[\texttt{com\_error}]$. The erroneous collaboration $\texttt{com\_error}$ 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 6.10, without loss of generality we can focus on the subterm $C_1 = (\bar{a}(x).P \mid a(y).Q)$ of C, such that $C_1 \rightarrowtail^* \big(\nu s : C_1\big)C'' \twoheadrightarrow \big(\nu s : C_1\big)\texttt{com\_error}$, with $C'' = (\langle\tilde{Q}_1\rangle \blacktriangleright P_1 \mid \langle\tilde{Q}_2\rangle \blacktriangleright P_2)$, and $C_1 \ \blacktriangleright\ \{\bar{a} : T_1, a : T_2\}$, with $T_1 \dashv\Vert T_2$.

By Lemma A.3, we have that there exist U_1, U_2, U_1', U_2', U_1'' such that $(T_1, T_2) : \langle T_1\rangle \blacktriangleright T_1 \parallel \langle T_2\rangle \blacktriangleright T_2 \longmapsto^* (T_1, T_2) : \langle\tilde{U}_1\rangle \blacktriangleright U_1' \parallel \langle\tilde{U}_2\rangle \blacktriangleright U_2' = t$ with $\tilde{Q}_1$ (resp. $\tilde{Q}_2$) in checkpoint accordance with $\tilde{U}_1$ (resp. $\tilde{U}_2$), $U_1' \xrightarrow{![S]} U_1''$, and for all U_2'' such that $U_2' \xrightarrow{\tau}{}^* U_2''$ we have $U_2'' \stackrel{?[S]}{\not\longrightarrow}$. Thus, for all U_2'' as above, we have $t \longmapsto^* (T_1, T_2) : \langle\tilde{U}_1\rangle \blacktriangleright U_1' \parallel \langle\tilde{U}_2\rangle \blacktriangleright U_2'' = t'$. No rule in Fig. 11 allows the term t' to evolve, i.e. $t' \not\longmapsto$, because U_1' can only perform $![S]$ and rule [TS-Com] cannot be applied since $U_2'' \stackrel{?[S]}{\not\longrightarrow}$. Since $T_1 \dashv\Vert T_2$, by Def. 4.1 it must hold that $U_1' = \texttt{end}$. However, since U_1' is able to perform an action (as it holds that $U_1' \xrightarrow{![S]} U_1''$), we get that it cannot be an end type, i.e. $U_1' \neq \texttt{end}$, which is a contradiction. □