The Reversible Temporal Process Language[*]

Laura Bocchi[1], Ivan Lanese[2], Claudio Antares Mezzina[3], and Shoji Yuen[4]

[1] School of Computing, University of Kent, UK
[2] Focus Team, University of Bologna/INRIA, Italy
[3] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy
[4] Nagoya University, Japan

Abstract. Reversible debuggers help programmers to quickly find the causes of misbehaviours in concurrent programs. These debuggers can be founded on the well-studied theory of causal-consistent reversibility, which allows one to undo any action provided that its consequences are undone beforehand. Till now, causal-consistent reversibility never considered time, a key aspect in real world applications. Here, we study the interplay between reversibility and time in concurrent systems via a process algebra. The Temporal Process Language (TPL) by Hennessy and Regan is a well-understood extension of CCS with discrete-time and a timeout operator. We define `revTPL`, a reversible extension of TPL, and we show that it satisfies the properties expected from a causal-consistent reversible calculus. We show that, alternatively, `revTPL` can be interpreted as an extension of reversible CCS with time.

1 Introduction

Recent studies [29,5] show that reversible debuggers ease the debugging phase, and help programmers to quickly find the causes of a misbehaviour. Reversible debuggers can be built on top of a causal-consistent reversible semantics [11,8], and this is particularly suited to deal with concurrency bugs, which are hard to find using traditional debuggers. By exploiting causality information, causal-consistent reversible debuggers allow one to undo just the steps which led (are causally related) to a misbehaviour, reducing the number of steps/spurious causes and helping to understand its root cause. In the last years several reversible semantics for concurrency have been developed, see, e.g., [7,6,17,2,21]. However, none of them takes into account time[1]. Time-dependent behaviour is an intrinsic and important feature of real-world concurrent systems and has

---

[1] The notion of time reversibility addressed by [2] is not aimed at studying hard or soft time constraints but at performance evaluation via (time-reversible) Markov chains.

many applications: from the engineering of highways [20], to the manufacturing schedule [10] and to the scheduling problem for real-time operating systems [3].

Time is instrumental for the functioning of embedded systems where some events are triggered by the system clock. Embedded systems are used for both real-time and soft real-time applications, frequently in safety-critical scenarios. Hence, before being deployed or massively produced, they have to be heavily tested. The testing activity may trigger a debugging phase: if a test fails one has to track down the source(s) of the failure and fix them. Actually, debugging occurs not only upon testing, but in almost all the stages of the life-cycle of a software system: from the early stages of prototyping to the post-release maintenance (e.g., updates or security patches). Concurrency is important in embedded systems [9], and concurrency bugs frequently happen in these systems as well [13]. To debug such systems, and deal with time-dependent bugs in particular, it is crucial that debuggers can handle concurrency and time.

In this paper, we study the interplay of time and reversibility in a process algebra for concurrency. There exists a variety of timed process algebras for the analysis and specification of concurrent timed systems [26]. We build on the Temporal Process Language (TPL) [12], a CCS-like process algebra featuring an *idling* prefix (modelling a delay) and a *timeout* operator. The choice of TPL is due to its simplicity and its well-understood theory. We define `revTPL`, a reversible extension of TPL, and we show that it satisfies the properties expected from a causal-consistent reversible calculus. Alternatively, `revTPL` can be interpreted as an extension of reversible CCS (in particular CCSK [28]) with time.

A reversible semantics in a concurrent setting is frequently defined following the causal-consistent approach [7] (other approaches are also used, e.g., to model biological systems [27]). Causal-consistent reversibility states that any action can be undone, provided that its consequences are undone beforehand. Hence, it strongly relies on a notion of causality. To prove the reversible semantics of `revTPL` causal-consistent, we exploit the theory in [19], whereby causal-consistency follows from three key properties: any action can be undone by a corresponding backward action (Loop Lemma); concurrent actions can be executed in any order (Square Property); backward computations do not introduce new states (Parabolic Lemma).

The application of causal-consistent reversibility to timed systems is not straightforward, since time heavily changes the causal semantics of the language. In untimed systems, causal dependencies are either *structural* (e.g., via sequential composition) or determined by *synchronisations*. In timed systems further dependencies between parallel processes can be introduced by time, even when processes do not actually interact, as illustrated in Example 1.

Example 1 (Motivating example). Consider the following Erlang code.

```
1  process_A () ->                    5        handleTimeout ()
2    receive                         6    end end.
3      X -> handleMsg ()
4      after 200 ->
```

```
7 process_B(Pid) ->              10 PidA=spawn(?MODULE, process_A ,[]),
8    timer:sleep(500),           11 spawn(?MODULE, process_B ,[PidA]).
9    Pid! Msg  end.
```

Two processes are supposed to communicate, but the timeout in process A (line 4) triggers after 200 ms, while process B will only send the message after 500 ms (lines $8-9$). In this example, the timeout is ruling out an execution that would be possible in the untimed scenario (the communication between A and B) and introduces a dependency without need of actual interactions. $\diamond$

From a technical point of view, the semantics of TPL does not fit the formats for which a causal-consistent reversible semantics can be built automatically [28,14], and also the generalisation of the approaches developed in the literature for untimed models [7,6,17] is not straightforward. Actually, we even need to change the underlying formalisation of TPL to ensure that its reversible extension is causal consistent (see Section 5.1).

The rest of the paper is structured as follows. Section 2 gives an informal overview of TPL and reversibility. Section 3 introduces the syntax and semantics of the reversible Temporal Process Language (`revTPL`). In Section 4, we relate `revTPL` to TPL and CCSK, while Section 5 studies the reversibility properties of `revTPL`. Section 6 concludes the paper. Proofs and additional technical details are collected in the associated technical report [?].

2   Informal overview of TPL and reversibility

In this section we give an informal overview of Hennessy & Regan's TPL (Temporal Process Language) [12] and introduce a few basic concepts of causal-consistent reversibility [7,19].

Overview of TPL. Process $\lfloor pid.P \rfloor(Q)$ models a timeout: it can either immediately do action pid followed by P or, in case of delay, continue as Q. In (1) the timeout process is in parallel with co-party $\overline{pid}.0$ that can immediately synchronise with action pid, and hence the timeout process continues as P.

$$\overline{pid}.0 \parallel \lfloor pid.P \rfloor(Q) \xrightarrow{\tau} 0 \parallel P \tag{1}$$

In (2), $\lfloor pid.P \rfloor(Q)$ is in parallel with process $\sigma.\overline{pid}.0$ that can synchronise only after a delay of one time unit σ (σ is called a time action). Because of the delay, the timeout process continues as Q:

$$\sigma.\overline{pid}.0 \parallel \lfloor pid.P \rfloor(Q) \xrightarrow{\sigma} \overline{pid}.0 \parallel Q \tag{2}$$

The processes in (2) describe the interaction structures of the Erlang program in Example 1. More precisely, the timeout of 200 time units in process A can be encoded using nested timeouts:

$$A(0) = Q \quad A(n+1) = \lfloor pid.P \rfloor(A(n)) \quad (n \in \mathbb{N})$$

while process B can be modelled as the sequential composition of 500 actions σ followed by action $\overline{pid}$, as follows:

$$B(0) = \overline{pid} \quad B(n+1) = \sigma.B(n) \quad (n \in \mathbb{N})$$

Using the definition above, $\lfloor pid.P \rfloor(A(200))$ models a process that executes pid and continues as P if a co-party is able to synchronise within 200 time units, otherwise executes Q. Hence, Example 1 is rendered as follows:

$$\lfloor pid.P \rfloor(A(200)) \parallel B(500)$$

The design of TPL is based on (and enjoys) three properties [12]: time-determinism, patience, and maximal progress. *Time-determinism* means that time actions from one state can never reach distinct states, formally: if $P \xrightarrow{\sigma} Q$ and $P \xrightarrow{\sigma} Q'$ then $Q = Q'$. A consequence of time-determinism is that choices can only be decided via communication actions and not by time actions, for example $\alpha.P + \beta.Q$ can change state by action α or β, but not by time action σ. Process $\alpha.P + \beta.Q$ can make an action σ, by a property called patience, but this action would not change the state, as shown in (3).

$$\alpha.P + \beta.Q \xrightarrow{\sigma} \alpha.P + \beta.Q \tag{3}$$

Patience ensures that communication processes $\alpha.P$ can indefinitely delay communication α with σ actions (without changing state) until a co-party is available. *Maximal progress* states that (internal/synchronisation) τ actions cannot be delayed, formally: if $P \xrightarrow{\tau} Q$ then $P \xrightarrow{\sigma} Q'$ for no Q'. Namely, a delay can only be attained via explicit σ prefixes or because synchronisation is not possible. Basically, patience allows for time actions when communication is not possible, and maximal progress disallows time actions when communication is possible:

$$\alpha.P \xrightarrow{\sigma} \qquad\qquad\qquad \text{(by patience)}$$
$$\alpha.P \parallel \overline{\alpha}.Q \not\xrightarrow{\sigma} \text{ because } \alpha.P \parallel \overline{\alpha}.Q \xrightarrow{\tau} \text{ (by maximal progress)}$$

Overview of causal-consistent reversibility. Before presenting `revTPL`, we discuss the *reversing technique* we adopt. In the literature, two approaches to define a causal-consistent extension of a given calculus or language have been proposed: dynamic and static [15]. The dynamic approach (as in [7,6,17]) makes explicit use of memories to keep track of past events and causality relations, while the static approach (originally proposed in [28]) is based on two ideas: making all the operators of the language static so that no information is lost and using communication keys to keep track of which events have been executed. In the dynamic approach, constructors of processes disappear upon reduction (as in standard calculi).

For example, in the following CCS reduction:

$$a.P \xrightarrow{a} P$$

the action a disappears as effect of the reduction. The dynamic approach prescribes to use a memory to keep track of the discarded items. In static approaches, such as [28], actions are syntactically maintained, and process $a.P$ reduces as follows

$$a.P \xrightarrow{a[i]} a[i].P$$

where P is decorated with the executed action a and a unique key i. The term $a[i].P$ acts like P in forward reductions, while the coloured part decorating P is used to define backward reductions, e.g.,

$$a[i].P \xleftarrow{a[i]} a.P$$

Keys are important to correctly revert synchronisations. Consider the process below. It can take two forward synchronisations with keys i and j, respectively:

$$a.P_1 \parallel \overline{a}.P_2 \parallel a.Q_1 \parallel \overline{a}.Q_2 \xrightarrow{\tau[i]}\xrightarrow{\tau[j]} a[i].P_1 \parallel \overline{a}[i].P_2 \parallel a[j].Q_1 \parallel \overline{a}[j].Q_2$$

From the reached state, there are two possible backward actions: $\tau[i]$ and $\tau[j]$. The keys are used to ensure that a backward action, say $\tau[i]$, only involves parallel components that have previously synchronised and not, for instance, $a[i].P_1$ and $\overline{a}[j].Q_2$. When looking at the choice operator, in the following CCS reduction:

$$a.P + b.Q \xrightarrow{a} P$$

both the choice operator "+" and the discarded branch $b.Q$ disappear as effect of the reduction. In static approaches, the choice operator and the discarded branch are syntactically maintained, and process $a.P + b.Q$ reduces as follows:

$$a.P + b.Q \xrightarrow{a[i]} a[i].P + b.Q$$

where $a[i].P + b.Q$ acts like P in forward reductions, while the coloured part allows one to undo $a[i]$ and then possibly proceed forward with an action $b[j]$.

In this paper, we adopt the static approach since it is simpler, while the dynamic approach is more suitable to complex languages such as the π-calculus, see the discussion in [15,18].

3   Reversible Temporal Process Language

In this section we define revTPL, an extension of TPL [12] with reversibility following the static approach in the style of [28].

Syntax of revTPL. We denote with $\mathcal{X}$ the set of all the processes generated by the grammar in Figure 1.

Programs $(P, Q, \ldots)$ describe timed interactions following [12]. We let $\mathcal{A}$ be the set of action names a, $\overline{\mathcal{A}}$ the set of action conames $\overline{a}$. We use α to range over a, $\overline{a}$ and internal actions τ. We assume $\overline{\overline{a}} = a$. In program $\pi.P$, prefix π can be a

$$P = \pi.P \mid \lfloor P \rfloor(Q) \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A \mid \mathbf{0} \qquad (\pi = \alpha \mid \sigma)$$

$$X = \pi[i].X \mid \lfloor X \rfloor[\underrightarrow{i}](Y) \mid \lfloor X \rfloor[\underleftarrow{i}](Y) \mid X + Y \mid X \parallel Y \mid X \setminus a \mid P$$

Fig. 1. Syntax of `revTPL`

communication action α or a time action σ, and P is the continuation. Timeout $\lfloor P \rfloor(Q)$ executes either P (if possible) or Q (in case of timeout). $P + Q$, $P \parallel Q$, $P \setminus a$, A, and $\mathbf{0}$ are the usual choice, parallel composition, name restriction, recursive call, and terminated program from CCS. For each recursive call A we assume a recursive definition $A \overset{def}{=} P$.

Processes $(X, Y, \ldots)$ describe states via annotation of executed actions with keys following the static approach. We let $\mathcal{K}$ be the set of all keys $(k, i, j, \ldots)$. Processes are programs with (possibly) some computational history (i.e., prefixes marked with keys): $\pi[i].X$ is the process that has already executed π, and the execution of such π is identified by key i. Process $\lfloor X \rfloor[\underleftarrow{i}](Y)$ is executing the main branch X whereas $\lfloor X \rfloor[\underrightarrow{i}](Y)$ is executing Y.

A process can be thought of as a context with actions that have already been executed, each associated to a key, containing a program P, with actions yet to execute and hence with no keys. Notably, keys are distinct but for actions happening together: an action and a co-action that synchronise, or the same timed action traced by different processes, e.g., by two parallel delays. A program P can be thought of as the initial state of a computation, where no action has been executed yet. We call such processes *standard*. Definition 1 formalises this notion via function $\texttt{keys}(X)$ that returns the set of keys of a process.

Definition 1 (Standard process). *The set of keys of a process X, written $\texttt{keys}(X)$, is inductively defined as follows:*

$$\texttt{keys}(P) = \emptyset \quad \texttt{keys}(\pi[i].X) = \{i\} \cup \texttt{keys}(X) \quad \texttt{keys}(X \setminus a) = \texttt{keys}(X)$$
$$\texttt{keys}(\lfloor Y \rfloor[\underrightarrow{i}](X)) = \texttt{keys}(\lfloor X \rfloor[\underleftarrow{i}](Y)) = \{i\} \cup \texttt{keys}(X)$$
$$\texttt{keys}(X + Y) = \texttt{keys}(X \parallel Y) = \texttt{keys}(X) \cup \texttt{keys}(Y)$$

A process X is standard, *written* $\texttt{std}(X)$, *if* $\texttt{keys}(X) = \emptyset$.

Basically, a standard process is a program. To handle the delicate interplay between time-determinism and reversibility of time actions, it is useful to distinguish the class of processes that have not executed any *communication* action (but may have executed time actions). We call these processes *not-acted* and characterise them formally using the predicate $\texttt{nact}(\cdot)$ below.

Definition 2 (Not-acted process). *The not-acted predicate* $\mathtt{nact}(\cdot)$ *is inductively defined as:*

$$\mathtt{nact}(\mathbf{0}) = \mathtt{nact}(A) = \mathtt{nact}(\lfloor X \rfloor(Y)) = \mathtt{nact}(\pi.X) = \mathtt{tt}$$
$$\mathtt{nact}(\alpha[i].X) = \mathtt{nact}(\lfloor X \rfloor \lfloor \underset{\leftarrow}{i} \rfloor(Y)) = \mathtt{ff}$$
$$\mathtt{nact}(\sigma[i].X) = \mathtt{nact}(X \setminus a) = \mathtt{nact}(\lfloor Y \rfloor \lfloor \underset{\rightarrow}{i} \rfloor(X)) = \mathtt{nact}(X)$$
$$\mathtt{nact}(X \parallel Y) = \mathtt{nact}(X + Y) = \mathtt{nact}(X) \wedge \mathtt{nact}(Y)$$

A process X is not-acted *(resp.* acted*) if* $\mathtt{nact}(X) = \mathtt{tt}$ *(resp.* $\mathtt{nact}(X) = \mathtt{ff}$*).*

Basic standard processes are always not-acted (first line of Definition 2). Indeed, it is not possible to reach a process $\pi.X$ where X is acted. In the second line, a process that has executed communication actions is acted. In particular, we will see that $\lfloor X \rfloor \lfloor \underset{\leftarrow}{i} \rfloor(Y)$ is only reachable via a communication action. The processes in the third line are not-acted if their continuations are not-acted. For parallel composition and choice, $\mathtt{nact}(\cdot)$ is defined as a conjunction. For example $\mathtt{nact}(\alpha[i].P \parallel \beta.Q) = \mathtt{ff}$ and $\mathtt{nact}(\alpha[i].P + \beta.Q) = \mathtt{ff}$. Note that in a choice process $X_1 + X_2$, at most one between X_1 and X_2 can be not-acted. Whereas $\mathtt{std}(X)$ implies $\mathtt{nact}(X)$, the opposite implication does not hold. For example, $\mathtt{std}(\sigma[i].\mathbf{0}) = \mathtt{ff}$ but $\mathtt{nact}(\sigma[i].\mathbf{0}) = \mathtt{tt}$.

Semantics of $\mathtt{revTPL}$**.** We denote with $\mathcal{A}^t$ the set $\mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau, \sigma\}$ of actions and let π to range over the set $\mathcal{A}^t$. We define the set of all the labels $\mathcal{L} = \mathcal{A}^t \times (\mathcal{K} \cup \{\star\})$. The labels associate each $\pi \in \mathcal{A}^t$ to either a key i or a wildcard $\star$. The key is used to associate the forward occurrence of an action with its corresponding reversal. Also, instances of actions occurring together (synchronising action and co-action or the effect of time passing in different components of a process) have the same key, otherwise keys are distinct. We introduce a wildcard $\star$ to label time transitions that leave the state unchanged. We call transitions with key $i \in \mathcal{K}$ *recorded* and transitions with $\star$ *patient*. We let $u, v, w, \ldots$ to range over $\mathcal{K} \cup \{\star\}$.

Definition 3 (Semantics). *The operational semantics of* $\mathtt{revTPL}$ *is given by two LTSs defined on the same set of all processes $\mathcal{X}$, and the set of all labels $\mathcal{L}$: a forward LTS $(\mathcal{X}, \mathcal{L}, \rightarrow)$ and a backward LTS $(\mathcal{X}, \mathcal{L}, \hookrightarrow)$. We define $\mapsto = \rightarrow \cup \hookrightarrow$, where $\rightarrow$ and $\hookrightarrow$ are the least transition relations induced by the rules in Figure 2 and Figure 3, respectively.*

Given a relation $\mathcal{R}$, we indicate with $\mathcal{R}^*$ its transitive and reflexive closure. We use notation $X \not\xrightarrow{\mathcal{I}}$ (resp. $X \not\xleftarrow{\mathcal{I}}$) for $X \xrightarrow{\mathcal{I}[i]} X'$ (resp. $X \xleftarrow{\mathcal{I}[i]} X'$) for any process X' and key i.

We now discuss the rules of the forward semantics (Figure 2). Rule [PAct] describes patient actions: program $\alpha.P$ can make a time step to itself. This kind of actions allows a process to wait indefinitely until it can communicate (by patience [12]). Since [PAct] does not change the state of $\alpha.P$, we do not track this action by associating it to wildcard $\star$ rather than to a key. [RAct]

$$\text{PACT } \alpha.P \xrightarrow{\sigma[\star]} \alpha.P \qquad \text{RACT } \pi.P \xrightarrow{\pi[i]} \pi[i].P \qquad \text{ACT } \dfrac{X \xrightarrow{\pi'[u]} X' \qquad u \neq i}{\pi[i].X \xrightarrow{\pi'[u]} \pi[i].X'}$$

$$\text{STOUT } \dfrac{X \not\xrightarrow{\bar{\tau}} \qquad \mathtt{std}(X) \qquad \mathtt{std}(Y)}{\lfloor X \rfloor(Y) \xrightarrow{\sigma[i]} \lfloor X \rfloor[\underrightarrow{i}](Y)} \qquad \text{SWAIT } \dfrac{Y \xrightarrow{\pi[u]} Y' \qquad u \neq i}{\lfloor X \rfloor[\underrightarrow{i}](Y) \xrightarrow{\pi[u]} \lfloor X \rfloor[\underrightarrow{i}](Y')}$$

$$\text{TOUT } \dfrac{X \xrightarrow{\alpha[i]} X' \qquad \mathtt{std}(Y)}{\lfloor X \rfloor(Y) \xrightarrow{\alpha[i]} \lfloor X' \rfloor[\underleftarrow{i}](Y)} \qquad \text{WAIT } \dfrac{X \xrightarrow{\pi[u]} X' \qquad u \neq i}{\lfloor X \rfloor[\underleftarrow{i}](Y) \xrightarrow{\pi[u]} \lfloor X' \rfloor[\underleftarrow{i}](Y)}$$

$$\text{SYNW } \dfrac{X \xrightarrow{\sigma[u]} X' \qquad Y \xrightarrow{\sigma[v]} Y' \qquad (X \parallel Y) \not\xrightarrow{\bar{\tau}} \qquad \delta(u,v) = w}{X \parallel Y \xrightarrow{\sigma[w]} X' \parallel Y'}$$

$$\text{PAR } \dfrac{X \xrightarrow{\alpha[i]} X' \qquad i \notin \mathtt{keys}(Y)}{X \parallel Y \xrightarrow{\alpha[i]} X' \parallel Y} \qquad \text{SYN } \dfrac{X \xrightarrow{\alpha[i]} X' \qquad Y \xrightarrow{\overline{\alpha}[i]} Y'}{X \parallel Y \xrightarrow{\tau[i]} X' \parallel Y'}$$

$$\text{CHOW1 } \dfrac{X_1 \xrightarrow{\sigma[u]} X_1' \qquad X_2 \xrightarrow{\sigma[v]} X_2' \qquad \delta(u,v) = w \qquad \mathtt{nact}(X_1 + X_2)}{X_1 + X_2 \xrightarrow{\sigma[w]} X_1' + X_2'}$$

$$\text{CHOW2 } \dfrac{X_1 \xrightarrow{\sigma[u]} X_1' \qquad \mathtt{nact}(X_2) \wedge \neg\mathtt{nact}(X_1)}{X_1 + X_2 \xrightarrow{\sigma[u]} X_1' + X_2} \qquad \text{CHO } \dfrac{X_1 \xrightarrow{\alpha[i]} X_1' \qquad \mathtt{nact}(X_2)}{X_1 + X_2 \xrightarrow{\alpha[i]} X_1' + X_2}$$

$$\text{IDLE } \mathbf{0} \xrightarrow{\sigma[\star]} \mathbf{0} \qquad \text{HIDE } \dfrac{X \xrightarrow{\pi[u]} X' \qquad \pi \notin \{a, \overline{a}\}}{X \setminus a \xrightarrow{\pi[u]} X' \setminus a} \qquad \text{CONST } \dfrac{A \stackrel{def}{=} P \qquad P \xrightarrow{\pi[u]} X}{A \xrightarrow{\pi[u]} X}$$

The set of rules also includes symmetric versions of rules [PAR], [CHOW2] and [CHO].

Fig. 2. `revTPL` forward LTS

executes recorded actions $\alpha[i]$ or $\sigma[i]$ on a prefix program. Observe that, unlike patient time actions on $\alpha.P$ (which may or may not happen depending on the context), a time action on $\sigma.P$ corresponds to a deliberate and planned time consuming action and is, therefore, recorded so that it may be later reversed. [ACT] lifts actions of the continuation X on processes where prefix $\pi[i]$ has already been executed. [STOUT] and [SWAIT] model timeouts. In [STOUT], if X is not able to make τ actions then Y is executed; this rule models a timeout that triggers only if the main process X is stuck. The negative premise on [STOUT] can be encoded into a decidable positive one as shown in the associated technical report [**?**]. In rule [TOUT] instead the main process can execute and the timeout does not trigger. Rule [SWAIT] (resp. [WAIT]) models transitions inside a timeout process where the Y (resp. X) branch has been previously taken. The semantics of timeout construct becomes clearer in the larger context of parallel processes, when looking at rule [SYNW]. Rule [SYNW] models time passing for parallel processes. The negative premise ensures that, in case X or Y is a timeout process, timeout can trigger only if no synchronisation may occur, that is if the processes are stuck. [SYNW] requires time to pass in the same way (an action σ is taken by both components) for the whole system. Note that u and v may or may not be wildcards, depending on the form of X and Y. To determine w we use a synchronisation function $\delta : (\mathcal{K} \cup \{\star\} \times \mathcal{K} \cup \{\star\}) \mapsto \mathcal{K} \cup \{\star\}$ defined as follows, assuming $i, j \in \mathcal{K}$:

$$\delta(i,i) = i \qquad \delta(i,\star) = \delta(\star,i) = i \qquad \delta(\star,\star) = \star \qquad \delta(i,j) = \bot \ (i \neq j)$$

Basically, in rule [SYNW], if either u or v needs to be recorded then w also needs to be recorded, and if both u and v need to be recorded then we require $u = v$. Rules [PAR] (and symmetric) and [SYN] are as usual for communication actions and allow parallel processes to either proceed independently or to synchronise. Defining the semantics of choice process $X_1 + X_2$ requires special care to ensure time-determinism (recall, choices are only decided via communication actions). Also, we need to record time actions (unless they have a wildcard) to be able to reverse them correctly (cfr. Loop Lemma, discussed later on in Lemma 1). Rule [CHOW1] is for time actions when no choice between X_1 and X_2 has been made yet (as enforced by premise $\texttt{nact}(X_1 + X_2)$), and a time action happens in both branches. As in rule [SYNW], w is determined by the synchronisation function $\delta(u,v)$. Rule [CHOW2] models a time action when branch X_1 has already been chosen, as enforced by premise $\neg\texttt{nact}(X_1)$; the time action only affects the 'active' branch X_1. For example, in the process below, the premise $\texttt{nact}(\beta.Q) \wedge \neg\texttt{nact}(\alpha[i].\sigma.P)$ allows us to apply [CHOW2] obtaining:

$$\alpha[i].\sigma.P + \beta.Q \xrightarrow{\sigma[j]} \alpha[i].\sigma[j].P + \beta.Q$$

Having rule [CHOW2], besides [CHOW1], for time actions on a chosen branch, is justified by scenarios as the following:

$$\alpha[i].\sigma.P + \lfloor Q \rfloor(R)$$

If we would model time transitions of the process above using [CHoW1] we would obtain

$$\alpha[i].\sigma.P + \lfloor Q \rfloor(R) \xrightarrow{\sigma[j]} \alpha[i].\sigma[j].P + \lfloor Q \rfloor[\underrightarrow{j}](R)$$

wrongly suggesting that the timeout on the right branch has evolved. Rule [CHo] allows one to take one branch, or continue executing a previously taken branch. The choice construct is syntactically preserved, to allow for reversibility, but the one branch that is not taken remains non-acted (i.e., $\texttt{nact}(X_2)$). This ensures that choices can be decided by a communication action only. Rules [IDLE], [HIDE], and [CONST] are standard, except that [IDLE] only does patient actions using wildcards, and [HIDE] and [CONST] may or may not use wildcards depending on the form of X.

The rules of the backward semantics, in Figure 3, undo actions previously recorded via the forward semantics, and they allow for timed actions with wildcard. Backward rules are symmetric to the forward ones.

Definition 4 (Initial and reachable processes). *A process X is initial if* $\texttt{std}(X)$. *A process X is reachable if it can be derived by an initial process.*

Basically, a process is initial if it has no computational history, and reachable if it can be obtained via forward and backward actions from an initial process.

4   Properties

We now give some properties of $\texttt{revTPL}$. In Section 4.1 we introduce a syntactic characterisation of the class of processes that can delay without changing state. In Section 4.2 we show that $\texttt{revTPL}$ extends both (non reversible) TPL and (untimed) reversible CCS.

4.1   Idempatience and properties of $\star$

We introduce a class of processes, which we call *idempatient*, that can make state-preserving patient actions. This class is key to define the causal-consistent reversible semantics of $\texttt{revTPL}$ (see Section 5.1).

Definition 5 (Idempatience). *We say that X is* idempatient *if* $\texttt{IP}(X)$ *where:*

$\texttt{IP}(\mathbf{0}) = \texttt{IP}(\alpha.P) = \texttt{tt} \qquad \texttt{IP}(\lfloor P \rfloor(Q)) = \texttt{IP}(\sigma.P) = \texttt{ff}$

$\texttt{IP}(X_1 \parallel X_2) = \texttt{IP}(X_1) \wedge \texttt{IP}(X_2) \wedge X_1 \parallel X_2 \overset{\tau}{\nrightarrow}$

$\texttt{IP}(X_1 + X_2) = \texttt{IP}(X_1) \wedge \texttt{IP}(X_2)$

$\texttt{IP}(\pi[i].X) = \texttt{IP}(\lfloor Y \rfloor[\underrightarrow{i}](X)) = \texttt{IP}(\lfloor X \rfloor[\underleftarrow{i}](Y)) = \texttt{IP}(X \setminus a) = \texttt{IP}(X)$

Proposition 1 shows that idempatience is a sound and complete characterisation of processes that can make state-preserving patient actions.

Proposition 1 (Idempatience). $\texttt{IP}(X) \iff X \overset{\sigma[\star]}{\longmapsto} X$.

$$\text{PACT } \alpha.P \xhookrightarrow{\sigma[\star]} \alpha.P \qquad \text{RACT } \pi[i].P \xhookrightarrow{\pi[i]} \pi.P \qquad \text{ACT } \dfrac{X \xhookrightarrow{\pi'[u]} X' \qquad u \neq i}{\pi[i].X \xhookrightarrow{\pi'[u]} \pi[i].X'}$$

$$\text{STOUT } \dfrac{X \xnrightarrow{\mathcal{I}} \qquad \mathtt{std}(X) \qquad \mathtt{std}(Y)}{\lfloor X \rfloor[\underrightarrow{i}](Y) \xhookrightarrow{\sigma[i]} \lfloor X \rfloor(Y)} \qquad \text{SWAIT } \dfrac{Y \xhookrightarrow{\pi[u]} Y' \qquad u \neq i}{\lfloor X \rfloor[\underrightarrow{i}](Y) \xhookrightarrow{\pi[u]} \lfloor X \rfloor[\underrightarrow{i}](Y')}$$

$$\text{TOUT } \dfrac{X \xhookrightarrow{\alpha[i]} X' \qquad \mathtt{std}(Y)}{\lfloor X \rfloor[\underleftarrow{i}](Y) \xhookrightarrow{\alpha[i]} \lfloor X' \rfloor(Y)} \qquad \text{WAIT } \dfrac{X \xhookrightarrow{\pi[u]} X' \qquad u \neq i}{\lfloor X \rfloor[\underleftarrow{i}](Y) \xhookrightarrow{\pi[u]} \lfloor X' \rfloor[\underleftarrow{i}](Y)}$$

$$\text{SYNW } \dfrac{X \xhookrightarrow{\sigma[u]} X' \qquad Y \xhookrightarrow{\sigma[v]} Y' \qquad (X \parallel Y) \xnrightarrow{\mathcal{I}} \qquad \delta(u,v) = w}{X \parallel Y \xhookrightarrow{\sigma[w]} X' \parallel Y'}$$

$$\text{PAR } \dfrac{X \xhookrightarrow{\alpha[i]} X' \qquad i \notin \mathtt{keys}(Y)}{X \parallel Y \xhookrightarrow{\alpha[i]} X' \parallel Y} \qquad \text{SYN } \dfrac{X \xhookrightarrow{\alpha[i]} X' \qquad Y \xhookrightarrow{\overline{\alpha}[i]} Y'}{X \parallel Y \xhookrightarrow{\tau[i]} X' \parallel Y'}$$

$$\text{CHOW1 } \dfrac{X_1 \xhookrightarrow{\sigma[u]} X_1' \qquad X_2 \xhookrightarrow{\sigma[v]} X_2' \qquad \delta(u,v) = w \qquad \mathtt{nact}(X_1 + X_2)}{X_1 + X_2 \xhookrightarrow{\sigma[w]} X_1' + X_2'}$$

$$\text{CHOW2 } \dfrac{X_1 \xhookrightarrow{\sigma[u]} X_1' \qquad \mathtt{nact}(X_2) \wedge \neg\mathtt{nact}(X_1)}{X_1 + X_2 \xhookrightarrow{\sigma[u]} X_1' + X_2} \qquad \text{CHO } \dfrac{X_1 \xhookrightarrow{\alpha[i]} X_1' \qquad \mathtt{nact}(X_2)}{X_1 + X_2 \xhookrightarrow{\alpha[i]} X_1' + X_2}$$

$$\text{IDLE } \mathbf{0} \xhookrightarrow{\sigma[\star]} \mathbf{0} \qquad \text{HIDE } \dfrac{X \xhookrightarrow{\pi[u]} X' \qquad \pi \notin \{a, \overline{a}\}}{X \setminus a \xhookrightarrow{\pi[u]} X' \setminus a} \qquad \text{CONST } \dfrac{A \stackrel{def}{=} P \qquad X \xhookrightarrow{\pi[u]} P}{X \xhookrightarrow{\pi[u]} A}$$

The set of rules also includes symmetric versions of rules [PAR], [CHOW2] and [CHO].

Fig. 3. `revTPL` backward LTS

Next, we give a property of time actions, and show that patient actions are state preserving.

Proposition 2 (Patient actions). $X \xrightarrow{\sigma[\star]}$ implies $X \xnrightarrow{\sigma[i]}$, and $X \xrightarrow{\sigma[i]}$ implies $X \xnrightarrow{\sigma[\star]}$. Moreover, if $X \xrightarrow{\sigma[\star]} Y$ then $X = Y$. The same for $\hookrightarrow$.

4.2 Relations with TPL and reversible CCS

We can consider `revTPL` as a reversible extension of TPL, but also as an extension of reversible CCS (in particular CCSK [28]) with time. First, if we consider the

forward semantics only, then we have a tight correspondence with TPL. To show this we define a forgetful map which discards the history information of a process.

Definition 6 (History forgetting map). *The history forgetting map* $\phi^{\mathtt{h}} : \mathcal{X} \to \mathcal{P}$ *is inductively defined as follows:*

$$\phi^{\mathtt{h}}(P) = P \qquad\qquad \phi^{\mathtt{h}}(\pi[i].X) = \phi^{\mathtt{h}}(X)$$

$$\phi^{\mathtt{h}}(\lfloor X \rfloor[\underleftarrow{i}](Y)) = \phi^{\mathtt{h}}(X) \qquad\qquad \phi^{\mathtt{h}}(\lfloor X \rfloor[\underrightarrow{i}](Y)) = \phi^{\mathtt{h}}(Y)$$

$$\phi^{\mathtt{h}}(X \parallel Y) = \phi^{\mathtt{h}}(X) \parallel \phi^{\mathtt{h}}(Y) \qquad\qquad \phi^{\mathtt{h}}(X \setminus a) = \phi^{\mathtt{h}}(X) \setminus a$$

$$\phi^{\mathtt{h}}(X_1 + X_2) = \begin{cases} \phi^{\mathtt{h}}(X_1) & \textit{if } \neg\mathtt{nact}(X_1) \wedge \mathtt{nact}(X_2) \\ \phi^{\mathtt{h}}(X_2) & \textit{if } \neg\mathtt{nact}(X_2) \wedge \mathtt{nact}(X_1) \\ \phi^{\mathtt{h}}(X_1) + \phi^{\mathtt{h}}(X_2) & \textit{otherwise} \end{cases}$$

In TPL time cannot decide choices. This is reflected into the definition of $\phi^{\mathtt{h}}(X_1 + X_2)$, where a branch disappears only if the other did an untimed action.

Notably, the restriction of $\phi^{\mathtt{h}}$ to untimed processes is a map from CCSK to CCS. In the following we will indicate with $\to_{\mathtt{t}}$ the semantics of TPL [12] and with $\mapsto_{\mathtt{k}}$ the semantics of CCSK [28].

Proposition 3 (Embedding of TPL). *Let X be a reachable* `revTPL` *process:*

1. *if $X \xrightarrow{\pi[u]} Y$ then $\phi^{\mathtt{h}}(X) \xrightarrow{\pi}_{\mathtt{t}} \phi^{\mathtt{h}}(Y)$;*
2. *if $\phi^{\mathtt{h}}(X) \xrightarrow{\pi}_{\mathtt{t}} Q$ then*

    – *either for any $i \in \mathcal{K} \setminus \mathtt{keys}(X)$ there is Y such that $X \xrightarrow{\pi[i]} Y$ or*

    – *$\pi = \sigma$ and there is Y such that $X \xrightarrow{\pi[\star]} Y$*

    In both the cases $\phi^{\mathtt{h}}(Y) = Q$.

Also, TPL is a conservative extension of CCS. This is stated in [12], even if not formally proved. Hence, we can define a *forgetful* map which discards all the temporal operators of a TPL term and get a CCS one. We can obtain a stronger result and relate `revTPL` with CCSK [28]. That is, if we consider the untimed part of `revTPL` what we get is a reversible CCS which is exactly CCSK. To this end, we define a time forgetting map $\phi^{\mathtt{t}}$. We denote with $\mathcal{X}^-$ the set of untimed reversible processes of `revTPL`. The set inclusion $\mathcal{X}^- \subset \mathcal{X}$ holds.

Definition 7 (Time forgetting map). *The time forgetting map $\phi^{\mathtt{t}} : \mathcal{X} \to \mathcal{X}^-$ is inductively defined as follows:*

$$
\begin{array}{ll}
\phi^{\mathtt{t}}(\mathbf{0}) = \mathbf{0} & \phi^{\mathtt{t}}(A) = A \\
\phi^{\mathtt{t}}(\alpha.P) = \alpha.\phi^{\mathtt{t}}(P) & \phi^{\mathtt{t}}(\alpha[i].X) = \alpha[i].\phi^{\mathtt{t}}(X) \\
\phi^{\mathtt{t}}(X + Y) = \phi^{\mathtt{t}}(X) + \phi^{\mathtt{t}}(Y) & \phi^{\mathtt{t}}(X \parallel Y) = \phi^{\mathtt{t}}(X) \parallel \phi^{\mathtt{t}}(Y) \\
\phi^{\mathtt{t}}(X \setminus a) = \phi^{\mathtt{t}}(X) \setminus a & \phi^{\mathtt{t}}(\lfloor X \rfloor(Y)) = \phi^{\mathtt{t}}(X) + \phi^{\mathtt{t}}(Y) \\
\phi^{\mathtt{t}}(\sigma.P) = \phi^{\mathtt{t}}(P) & \phi^{\mathtt{t}}(\sigma[i].X) = \phi^{\mathtt{t}}(X) \\
\phi^{\mathtt{t}}(\lfloor X \rfloor[\underleftarrow{i}](Y)) = \phi^{\mathtt{t}}(X) + \phi^{\mathtt{t}}(Y) & \phi^{\mathtt{t}}(\lfloor X \rfloor[\underrightarrow{i}](Y)) = \phi^{\mathtt{t}}(X) + \phi^{\mathtt{t}}(Y)
\end{array}
$$

Notably, the restriction of $\phi^{\mathtt{t}}$ to standard processes is a map from TPL to CCS.

The most interesting aspect in the definition above is that the temporal operator $\lfloor X \rfloor(Y)$ is rendered as a sum. This also happens for the decorated processes $\lfloor X \rfloor [\underleftarrow{i}](Y)$ and $\lfloor X \rfloor [\underrightarrow{i}](Y)$. Also, since we are relating a temporal semantics with an untimed one (CCSK), the σ actions performed by the timed semantics are not reflected in CCSK.

Proposition 4 (Embedding of CCSK [28]). *Let X be a reachable* `revTPL` *process. We have:*

1. *if $X \xmapsto{\alpha[i]} Y$ then $\phi^{\mathtt{t}}(X) \xmapsto{\alpha[i]}_{\mathtt{k}} \phi^{\mathtt{t}}(Y)$;*
2. *if $X \xmapsto{\sigma[u]} Y$ then $\phi^{\mathtt{t}}(X) = \phi^{\mathtt{t}}(Y)$;*

Notably, it is not always the case that transitions of the underlying untimed process can be matched in a timed setting, think, e.g., to the process in Example 1 (and its formalisation in Section 2) for a counterexample.
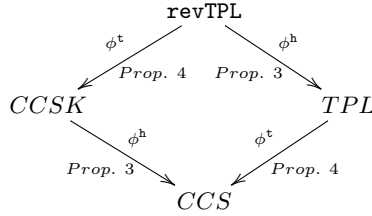

Fig. 4. Forgetting maps.

Figure 4 summarises our results: if we remove the timed behaviour from a `revTPL` process we get a CCSK term, thanks to Proposition 4. On the other side, if from `revTPL` we remove reversibility we get a TPL term (thanks to Proposition 3). Note that the same forgetful maps (and properties) justify the arrows in the bottom part of the diagram, as discussed above. This is in line with Theorem 5.21 of [28], showing that by removing reversibility and history information from CCSK we get CCS.

5 Reversibility in `revTPL`

In a fully reversible calculus any computation can be undone. This is a fundamental property of reversibility [7,19], and `revTPL` enjoys it. Formally:

Lemma 1 (Loop Lemma). *If X is a reachable process, then $X \xrightarrow{\pi[u]} X' \iff X' \xleftarrow{\pi[u]} X$*

Another fundamental property of causal-consistent reversibility is causal-consitency [7,19], which essentially states that we store the correct amount of causal information. In order to discuss it, we now borrow some definitions from [7]. We use t, t', s, s' to range over transitions. In a transition $t : X \xmapsto{\pi[u]} Y$ we call X the *source* of the transition, and Y the *target* of the transition. Two transitions are said to be *coinitial* if they have the same source, and *cofinal* if they have the same target. Given a transition t, we indicate with $\underline{t}$ its opposite, that is if $t : X \xmapsto{\pi[u]} Y$ (resp., $t : X \xhookleftarrow{\pi[u]} Y$) then $\underline{t} : Y \xhookleftarrow{\pi[u]} X$ (resp., $\underline{t} : Y \xmapsto{\pi[u]} X$). We let ρ, ω to range over sequences of transitions, which we call *paths*, and with ϵ_X we indicate the empty sequence starting and ending in X.

Definition 8 (Causal Equivalence). *Let $\asymp$ be the smallest equivalence on paths closed under composition and satisfying:*

1. *if $t : X \xmapsto{\pi_1[u]} Y_1$ and $s : X \xmapsto{\pi_2[v]} Y_2$ are independent, and $s' : Y_1 \xmapsto{\pi_2[v]} Z$, $t' : Y_2 \xmapsto{\pi_1[u]} Z$ then $ts' \asymp st'$;*
2. *$t\underline{t} \asymp \epsilon$ and $\underline{t}t \asymp \epsilon$*

Intuitively, paths are causal equivalent if they differ only for swapping independent transitions (we will discuss independence below) and for adding do-undo or undo-redo pairs of transitions.

Definition 9 (Causal Consistency (CC)). *If ρ and ω are coinitial and cofinal paths then $\rho \asymp \omega$.*

Intuitively, if coinitial paths are cofinal then they have the same causal information and can reverse in the same ways: we want only causal equivalent paths to reverse in the same ways.

Unfortunately, causal consistency does not hold in `revTPL` as defined in the previous sections (and this is not related to a specific definition of independence). This is due to actions with label $\sigma[\star]$.

Example 2 (CC does not hold with $\sigma[\star]$). Consider the path $\rho : \alpha.P \xrightarrow{\sigma[\star]} \alpha.P$. Trivially, ρ and $\epsilon_{\alpha.P}$ are coinitial and cofinal, but not causal equivalent. Indeed, the number of forward transitions minus the number of backward transitions is invariant under causal equivalence, and this is not the same for the two paths.

$\diamond$

This leaves us two possibilities to enforce the property: either we change the definition of causal equivalence (e.g., allowing one to freely add and remove transitions with label $\sigma[\star]$), or we change the semantics. We opt for the latter, since it allows us to stay in the framework studied in [19] and exploit the theory developped there to prove our results.

5.1 Revised semantics

We change the semantics simply dropping all transitions with label $\sigma[\star]$. Technically, this ensures that causal consistency (as well as other relevant properties) holds. Conceptually, those transitions do not amount to actual actions of the process (as shown in Proposition 2, they do not change the process) and are mainly used to simplify a compositional definition of the semantics, see, e.g., rule [SynW]. A compositional semantics could be defined by replacing premises of the form $X \xrightarrow{\sigma[\star]} X$ with $\texttt{IP}(X)$ thanks to Proposition 1. This option is discussed in more detail in the companion technical report [?].

For simplicity, from now on we consider the semantics given by the labelled transition system obtained by dropping all transitions with label $\sigma[\star]$:

Definition 10 (Semantics with no self-loops). *The operational semantics with no self-loops of* revTPL *is given by the forward LTS* $(\mathcal{X}, \mathcal{L}, \rightarrow_n)$ *and the backward LTS* $(\mathcal{X}, \mathcal{L}, \hookrightarrow_n)$*, on the same sets* $\mathcal{X}$ *of processes and* $\mathcal{L}$ *of labels. Transition relations* $\rightarrow_n$ *and* $\hookrightarrow_n$ *are obtained by dropping from, respectively,* $\rightarrow$ *and* $\hookrightarrow$ *the transitions with label* $\sigma[\star]$*.*

Notably, the Loop Lemma holds also for the new semantics. Concerning embeddings, the embedding of TPL does not hold any more (a new operational correspondence taking care of dropped transitions can be defined), while the one of CCSK is unaffected.

We discuss below reversibility in revTPL using the semantics with no self-loops. We first need to discuss the notion of independence.

5.2 Independence

We now define a notion of independence between revTPL transitions, based on a causality preorder (inspired by [18]) on keys. Independence is useful to show that reversibility never breaks causal links between actions.

Definition 11 (Partial order on keys). *The function* $\texttt{po}(\cdot)$*, that computes the set of causal relations among the keys in a process, is inductively defined as:*

$$\texttt{po}(P) = \emptyset \qquad \texttt{po}(X \setminus a) = \texttt{po}(X)$$
$$\texttt{po}(X \parallel Y) = \texttt{po}(X + Y) = \texttt{po}(\lfloor X \rfloor(Y)) = \texttt{po}(X) \cup \texttt{po}(Y)$$
$$\texttt{po}(\pi[i].X) = \texttt{po}(\lfloor X \rfloor[\underset{\leftarrow}{i}](Y)) = \texttt{po}(\lfloor Y \rfloor[\underset{\rightarrow}{i}](X)) = \{i < j \mid j \in \texttt{keys}(X)\} \cup \texttt{po}(X)$$

The partial order $\leq_X$ *on* $\texttt{keys}(X)$ *is the reflexive and transitive closure of* $\texttt{po}(X)$*.*

Let us note that function po computes a partial order relation, namely a set of pairs (i, j), denoted $i < j$ to stress that they form a partial order.

Definition 12 (Choice context). *A choice context* $\mathbf{C}$ *is a process with a hole* $\bullet$ *defined by the following grammar (we omit symmetric cases for* $+$ *and* $\parallel$*):*

$$\mathbf{C} = \bullet \mid \pi[i].\mathbf{C} \mid \lfloor \mathbf{C} \rfloor(Y) \mid \lfloor X \rfloor[\underset{\rightarrow}{i}](\mathbf{C}) \mid \lfloor \mathbf{C} \rfloor[\underset{\leftarrow}{i}](Y) \mid X + \mathbf{C} \mid X \parallel \mathbf{C} \mid \mathbf{C} \setminus a$$

Intuitively, a choice context may enclose an enabled (forward) choice. We now define a notion of conflict, and independence as its negation. For simplicity of formalisation, we assume that generation of fresh keys in forward transitions is deterministic: the same redex in the same process cannot generate different keys.

Definition 13 (Conflict and independence). *Given a reachable process X, two coinititial transitions $t : X \xmapsto{\pi_1[i]}_n Y$ and $s : X \xmapsto{\pi_2[j]}_n Z$ are conflicting, written $t \,\#\, s$, if and only if one of the following conditions holds:*

1. *$X \xrightarrow{\sigma[i]}_n Y$ and $X \xrightarrow{\alpha[j]}_n Z$;*
2. *$X \xrightarrow{\pi_1[i]}_n Y$ and $X \xleftarrow{\pi_2[j]}_n Z$ with $j \leq_Y i$;*
3. *$X = \mathbf{C}[Y' + Z'], Y' \xrightarrow{\pi_1[i]}_n Y''$ and $Z' \xrightarrow{\pi_2[j]}_n Z''$.*

Transitions t and s are independent*, written $t \,\mathcal{I}\, s$, if $t \neq s$ and they are not conflicting.*

The first clause tells us that a delay cannot be swapped with a communication action. Consider process $\lfloor b.\mathbf{0} \rfloor(\mathbf{0})$:
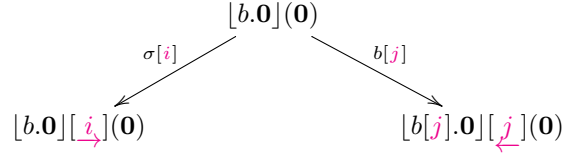
Transitions $\sigma[i]$ and $b[j]$ are in conflict: they cannot be swapped since action b is no longer possible after action σ, and vice versa. The second clause dictates that two transitions are in conflict when a reverse step eliminates some causes of a forward step. E.g., process $a[i].b.\mathbf{0}$ can do a forward step with label $b[j]$ or a backward one with label $a[i]$. Undoing $a[i]$ disables the action on b. The last case is the most intuitive: processes in different branches of a choice operator are in conflict, e.g., $a.\mathbf{0} + b.\mathbf{0}$ can do actions on a and b, but they can not be swapped.

The Square Property tells that two coinitial independent transitions commute, thus closing a diamond. Formally:

Property 1 (Square Property - SP) *Given a reachable process X and two coinititial transitions $t : X \xmapsto{\pi_1[i]}_n Y$ and $s : X \xmapsto{\pi_2[j]}_n Z$ with $t \,\mathcal{I}\, s$ there exist two cofinal transitions $t' : Y \xmapsto{\pi_2[j]}_n W$ and $s' : Z \xmapsto{\pi_1[i]}_n W$.*

5.3 Causal consistency

We can now prove causal consistency, using the theory in [19]. It ensures that causal consistency follows from SP, already discussed, and two other properties,

stated below. BTI (Backward Transitions are Independent) generalises the concept of backward determinism used for reversible sequential languages [30]. It specifies that two backward transitions from a same process are always independent.

Property 2 (Backward transition are independent - BTI) *Given a reachable process X, any two distinct coinitial backward transitions $t : X \xleftarrow{\pi_1[i]}_n Y$ and $s : X \xleftarrow{\pi_2[j]}_n Z$ are independent.*

The property trivially holds since by looking at the definition on conflicting and independent transitions (Definition 13) there are no cases in which two backward transitions are deemed as conflicting, hence two backward transitions are always independent.

We now show that reachable processes have a finite past.

Property 3 (Well-Foundedness - WF) *Let X_0 be a reachable process. Then there is no infinite sequence such that $X_i \xleftarrow{\pi_n[j_n]}_n X_{i+1}$ for all $i = 0, 1, \ldots$.*

WF follows since each backward transition removes a key.

The following lemma tells us that any path is causally equivalent to a path made by only backward steps, followed by only forward steps. In other words, up to causal equivalence, paths can be rearranged so as to first reach the maximum freedom of choice, going only backwards, and then continuing only forwards.

Definition 14 (Parabolic Lemma). *For any path ρ, there exist two forward-only paths ω, ω' such that $\rho \asymp \underline{\omega}\omega'$ and $|\omega| + |\omega'| \leq |\rho|$.*

We can now prove our main results thanks to the proof schema of [19].

Theorem 1 (From [19]). *Suppose BTI and SP hold, then PL holds. Suppose WF and PL hold, then CC holds.*

6 Conclusions

The main contribution of this paper is the study of the interplay between time and causal-consistent reversibility. A reversible semantics for TPL cannot be automatically derived using well-established frameworks [28,14], since some operator acts differently depending on whether the label is a communication or a time action. For example, in TPL a choice cannot be decided by the passage of time, making the $+$ operator both static and dynamic, and the approach in [28] not applicable. To faithfully capture patient actions in a reversible semantics we introduced wildcards. However, as $\sigma[\star]$ actions violate causal consistency, to recover it we had to refine the formalisation of the semantics. Another peculiarity of TPL is the timeout operator $\lfloor P \rfloor(Q)$, which can be seen as a choice operator whose left branch has priority over the right one. Although we have been able to use the static approach to reversibility [28], adapting it to our setting has been

challenging for the aforementioned reasons. Notably, our results have a double interpretation: as an extension of CCSK [28] with time, and as a reversible extension of TPL [12]. As a side result, by focusing on the two fragments, we derive notions of independence and conflict for CCSK and TPL, which were not available in the literature. We have just started to study the relations among revTPL, CCSK and TPL. We leave as future work a further investigation in terms of behavioural equivalences or simulations among the three calculi.

Maximal progress of TPL (as well as `revTPL`) has connections with Markov chains [4], e.g., $\tau.P + (\lambda).Q$ (where λ is a rate) will not be delayed since τ is instantaneously enabled. This resembles maximal progress for the timeout operator. A deep comparison between deterministic time, used by TPL, and stochastic time used by stochastic process algebras can be found in [1]. Further investigation on the relation between our work and [2], studying reversibility in Markov chains, is left for future work. The treatment of passage of time shares some similarities with broadcast [23]: time actions affect parallel components in the same way, and idempatience can be seen as unavailability of top-level receivers.

We have just started our research quest towards a reversible timed semantics. A further improvement would be to add an explicit rollback operator, as in [16], that could be triggered, e.g., in reaction to a timeout. Also, asynchronous communications (like in Erlang) could be taken into account. TPL is a conservative timed extension of CCS. Due to its simplicity, it has a very clear behavioural theory [12]. A further step could be to adapt such behavioural theory to account for reversibility. Also, we could consider studying more complex temporal operators [26]. Timed Petri nets are a valid tool for analysing real-time systems. A step towards the analysis of real-time systems would be to encode `revTPL` into (reversible) timed Petri nets [31], by extending the encoding of reversible CCS into reversible Petri nets [22]. Another possibility would be to study the extension of a monitored timed semantics for multiparty session types, as the one of [25], with reversibility [24].

References

1. M. Bernardo, F. Corradini, and L. Tesei. Timed process calculi with deterministic or stochastic delays: Commuting between durational and durationless actions. *Theor. Comput. Sci.*, 629:2–39, 2016.

2. M. Bernardo and C. A. Mezzina. Towards bridging time and causal reversibility. In A. Gotsman and A. Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems, FORTE 2020*, volume 12136 of *Lecture Notes in Computer Science*, pages 22–38. Springer, 2020.

3. I. C. Bertolotti. Real-time embedded operating systems. In R. Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005.

4. E. Brinksma and H. Hermanns. Process algebra and markov chains. In E. Brinksma, H. Hermanns, and J. Katoen, editors, *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 183–231. Springer, 2000.

5. T. Britton, L. Jeng, G. Carver, and P. Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers".

6. I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible π-calculus. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 388–397. IEEE Computer Society, 2013.

7. V. Danos and J. Krivine. Reversible communicating systems. In P. Gardner and N. Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

8. G. Fabbretti, I. Lanese, and J.-B. Stefani. Causal-Consistent Debugging of Distributed Erlang Programs. In S. Yamashita and T. Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2021.

9. J. S. Fant, H. Gomaa, and R. G. Pettit IV. A comparison of executable model based approaches for embedded systems. In H. Zhang, L. Zhu, and I. Kuz, editors, *Second International Workshop on Software Engineering for Embedded Systems, SEES 2012*, pages 16–22. IEEE, 2012.

10. M. Ghaleb, H. Zolfagharinia, and S. Taghipour. Real-time production scheduling in the industry-4.0 context: Addressing uncertainties in job arrivals and machine breakdowns. *Computers & Operations Research*, 123:105031, 2020.

11. E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In S. Gnesi and A. 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.

12. M. Hennessy and T. Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995.

13. P. Koopman. *Better Embedded System Software*. Drumnadrochit Press, 2010.

14. I. Lanese and D. Medic. A general approach to derive uncontrolled reversible semantics. In I. Konnov and L. Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020*, volume 171 of *LIPIcs*, pages 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

15. I. Lanese, D. Medic, and C. A. Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, 58(1-2):1–34, 2021.

16. I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In J. Katoen and B. König, editors, *CONCUR 2011 - Concurrency Theory - 22nd International Conference*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

17. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

18. I. Lanese and I. Phillips. Forward-reverse observational equivalences in CCSK. In S. Yamashita and T. Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings*, volume 12805 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2021.

19. I. Lanese, I. C. C. Phillips, and I. Ulidowski. An axiomatic approach to reversible computation. In J. Goubault-Larrecq and B. König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOS-SACS 2020*, volume 12077 of *Lecture Notes in Computer Science*, pages 442–461. Springer, 2020.

20. R. Mauro and A. Pompigna. State of the art and computational aspects of time-dependent waiting models for non-signalised intersections. *Journal of Traffic and Transportation Engineering (English Edition)*, 7(6):808–831, 2020.

21. D. Medic, C. A. Mezzina, I. Phillips, and N. Yoshida. A parametric framework for reversible π-calculi. *Inf. Comput.*, 275:104644, 2020.

22. H. C. Melgratti, C. A. Mezzina, and G. M. Pinna. Towards a truly concurrent semantics for reversible CCS. In S. Yamashita and T. Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2021.

23. C. A. Mezzina. On reversibility and broadcast. In J. Kari and I. Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018*, volume 11106 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2018.

24. C. A. Mezzina and J. A. Pèrez. Causal Consistency for Reversible Multiparty Protocols. *Logical Methods in Computer Science*, Volume 17, Issue 4, Oct. 2021.

25. R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017.

26. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K. G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer, 1991.

27. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In R. Glück and T. Yokoyama, editors, *Reversible Computation, 4th International Workshop, RC 2012*, volume 7581 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2012.

28. I. C. C. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007.

29. M. Vizard. Report: Debugging efforts cost companies \$61b annually, 2020.

30. T. Yokoyama and R. Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and E. Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 144–153. ACM, 2007.

31. A. Zimmermann, J. Freiheit, and G. Hommel. Discrete time stochastic petri nets for the modeling and evaluation of real-time systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, page 100. IEEE Computer Society, 2001.