

revTPL: THE REVERSIBLE TEMPORAL PROCESS LANGUAGE^{*}

LAURA BOCCHI ^a, IVAN LANESE ^b, CLAUDIO ANTARES MEZZINA ^c, AND SHOJI YUEN ^d

^a School of Computing, University of Kent, UK
e-mail address: L.Bocchi@kent.ac.uk

^b Focus Team, University of Bologna/INRIA, Italy
e-mail address: ivan.lanese@gmail.com

^c Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy
e-mail address: claudio.mezzina@uniurb.it

^d Graduate School of Informatics, Nagoya University, Japan
e-mail address: yuen@i.nagoya-u.ac.jp

ABSTRACT. Reversible debuggers help programmers to find the causes of misbehaviours in concurrent programs more quickly, by executing a program backwards from the point where a misbehaviour was observed, and looking for the bug(s) that caused it. Reversible debuggers can be founded on the well-studied theory of causal-consistent reversibility, which only allows one to undo an action provided that its consequences, if any, are undone beforehand. Causal-consistent reversibility yields more efficient debugging by reducing the number of states to be explored when looking backwards. Till now, causal-consistent reversibility has never considered time, which is 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.

Key words and phrases: Reversible computing; timed systems; process calculi; operational semantics.

^{*} This paper is a revised and extended version of [BLMY22].

This work has been partially supported by the BehAPI project funded by the EU H2020 RISE under the Marie Skłodowska-Curie action (No: 778233), by the EU HEU Marie Skłodowska-Curie action ReGraDe-CS (No: 101106046), by EPSRC project EP/T014512/1 (STARDUST), by MIUR PRIN project NiRvAna, by MIUR PRIN project DeKLA, by French ANR project DCore ANR-18-CE25-0007, by INdAM – GNCS 2022 project *Proprietà qualitative e quantitative di sistemi reversibili* and GNCS 2023 project *Reversibilità In Sistemi COncorrenti: analisi quantitative e funzionali (RISICO)*, code CUP_E53C22001930001, and by JSPS KAKENHI Grant Number JP21H03415. We thank the anonymous referees of this paper and of its conference version for their helpful comments and suggestions.

INTRODUCTION

Recent studies [Viz20, BJCC13] 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 [GLM14, LNPV18, FLS21, LLS⁺22], and this approach is particularly suited to deal with concurrency bugs, which are hard to find using traditional debuggers [Gra86]. By exploiting causality information, causal-consistent reversible debuggers allow one to undo just the steps which led (that is, are causally related) to a visible misbehaviour, reducing the number of steps/spurious causes and helping to understand the root cause of the misbehaviour. More precisely, one can explore backwards the tree of causes of a visible misbehaviour, possibly spread among different processes, looking for the bug(s) causing it. In the last years several reversible semantics for concurrency have been developed, see, e.g., [DK04, PU07, CKV13, LMS16, MMU19, GLMT17, LM20, BM20, MMPY20, MMP21a]. However, none of them takes into account time¹. Time-dependent behaviour is an intrinsic and important feature of real-world concurrent systems and has many applications: from the engineering of highways [MP20], to the manufacturing schedule [GZT20] and to the scheduling problem for real-time operating systems [Ber05].

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, and hence possibly debugged. 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 [FGP12], and concurrency bugs frequently happen in these systems as well [Koo10]. To debug such systems, and deal with time-dependent bugs in particular, it is crucial that debuggers can handle both concurrency and time.

In this paper, we study the interplay between time and reversibility in a process algebra for concurrency. In the literature, there exists a variety of timed process algebras for the analysis and specification of concurrent timed systems [NS91]. We build on the Temporal Process Language (TPL) [HR95], 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 [PU07]) with time.

A reversible semantics in a concurrent setting is frequently defined following the causal-consistent approach [DK04, LMT14] (other approaches are also used, e.g., to model biological systems [PUY12, PP18]). Causal-consistent reversibility states that any action can be undone, provided that its consequences, if any, 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 [LPU20], whereby causal-consistency follows from three key properties:

Loop Lemma: any action can be undone by a corresponding backward action;

¹The notion of time reversibility addressed in [BM20] is not aimed at studying programming languages with constructs to support hard or soft time constraints, but at performance evaluation via (time-reversible) Markov chains.

Square Property: concurrent actions can be executed in any order;

Parabolic Lemma: backward computations do not introduce new states.

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 the following example.

Example 0.1 (Motivating example). Consider the following Erlang code.

```

1 process_A () ->
2   receive
3     X -> handleMsg ()
4     after 200 ->
5       handleTimeout ()
6   end
7 end.
8 process_B (Pid) ->
9   timer:sleep (500) ,
10  Pid! Msg
11 end.
12
13 PidA=spawn (?MODULE, process_A , [] ) ,
14 spawn (?MODULE, process_B , [PidA] ) .

```

Process A (lines 1 – 7) waits for a message; if a message is received within 200 ms, then process A calls function `handleMsg()`, otherwise it calls function `handleTimeout()`. Process B (lines 8 – 11) sleeps for 500ms and then sends a message to `Pid`, where `Pid` is a parameter of the function executed by process B (line 8). The code in line 13 spawns an instance of process A and uses its process identifier `PidA` as a parameter to spawn an instance of process B (line 14). The two process instances 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 9 – 10). In this example, the timeout rules out the execution where process A communicates with process B, which would be possible in the untimed scenario. Namely, an execution can become unviable because of a time dependency, without any actual interaction between the two involved processes. \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 derived automatically [PU07, LM20], and also the generalisation of the approaches developed in the literature for untimed models [DK04, CKV13, LMS16] is not straightforward and is the objective of this work.

The rest of the paper is structured as follows. Section 1 gives an informal overview of TPL and reversibility. Section 2 introduces the syntax and semantics of the reversible Temporal Process Language (**revTPL**). In Section 3, we relate **revTPL** to TPL and CCSK, while Section 4 studies the reversibility properties of **revTPL**. Section 5 concludes the paper and discusses related and future work. A formal background on CCS, TPL and CCSK (for the readers that wish a more rigorous overview than the informal one in Section 1), as well as longer proofs and additional technical details are collected in Appendix.

This paper is an extended and revised version of [BLMY22]. The semantics has been revised since the one in [BLMY22] failed to capture some time dependencies when going back and forward (cf. Example 4.19). We now also provide a better characterisation of the causality model of **revTPL** (cf. Proposition 4.10 and Theorem 4.18). Further technical improvements include the formulation of the correspondences between **revTPL**, TPL, CCSK

and CCS in terms of (bi)simulations (Theorems 3.4 and 3.8). We now also provide full proofs of all results as well as additional explanations and examples. Finally, the whole presentation has been carefully refined.

1. INFORMAL OVERVIEW OF TPL AND REVERSIBILITY

In this section we give an informal overview of Hennessy & Regan's TPL (Temporal Process Language) [HR95] and introduce a few basic concepts of causal-consistent reversibility [DK04, LPU20]. For a more rigorous introduction, the interested reader can find the syntax and semantics of TPL in Appendix A.3 and the syntax and semantics of the reversible calculus CCSK [PU07] in Appendix A.2. The syntax and semantics of CCS, which is at the basis of both TPL and CCSK, is in Appendix A.1.

1.1. 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 transition (1.1) 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 .

$$\overline{pid}.0 \parallel \lfloor pid.P \rfloor(Q) \xrightarrow{\tau} 0 \parallel P \quad (1.1)$$

In transition (1.2), $\lfloor pid.P \rfloor(Q)$ 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 :

$$\sigma.\overline{pid}.0 \parallel \lfloor pid.P \rfloor(Q) \xrightarrow{\sigma} \overline{pid}.0 \parallel Q \quad (1.2)$$

The processes on the left-hand side of transition (1.2) describe the interaction structures of the Erlang program in Example 0.1. 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 0.1 is rendered as follows:

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

The design of TPL is based on (and enjoys) three properties [HR95]: 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 transition (1.3).

$$\alpha.P + \beta.Q \xrightarrow{\sigma} \alpha.P + \beta.Q \quad (1.3)$$

Patience ensures that communication processes like $\alpha.P$ can indefinitely delay communication α with σ actions (without changing state) until a co-party is available. For example,

by patience, process $\alpha.P$ in (1.4) can delay the communication on α until the other process $\sigma.\bar{\alpha}.Q$ is ready to communicate:

$$\alpha.P \parallel \sigma.\bar{\alpha}.Q \xrightarrow{\sigma} \alpha.P \parallel \bar{\alpha}.Q \xrightarrow{\tau} P \parallel Q \quad (1.4)$$

Maximal progress states that (internal/synchronisation) τ actions cannot be delayed, formally: if $P \xrightarrow{\tau} Q$ then there is no Q' such that $P \xrightarrow{\sigma} Q'$. Namely, a delay can only be attained either 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:

$$\begin{array}{ll} \alpha.P \xrightarrow{\sigma} & \text{(by patience)} \\ \alpha.P \parallel \bar{\alpha}.Q \not\xrightarrow{\tau} & \text{because } \alpha.P \parallel \bar{\alpha}.Q \xrightarrow{\tau} \text{ (by maximal progress)} \end{array}$$

1.2. 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* [LMM21]. The dynamic approach (as in [DK04, CKV13, LMS16]) makes explicit use of memories to keep track of past events and causality relations, while the static approach (originally proposed in [PU07]) 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 transitions (as in standard calculi).

For example, in the following CCS transition:

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

the action a disappears as an effect of the transition. The dynamic approach prescribes to use memories to keep track of the discarded items. In static approaches, such as [PU07], actions are syntactically maintained, and process $a.P$ can perform the transition below

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

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

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

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

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

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 $\bar{a}[j].Q_2$. When looking at the choice operator, in the following CCS transition:

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

$$\begin{aligned}
(\text{Processes}) \quad & P = \pi.P \mid [P](Q) \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A \mid \mathbf{0} \\
(\text{Configurations}) \quad & X = \rho[i].X \mid [X][\underline{i}](Y) \mid [X][\underline{j}](Y) \mid X + Y \mid \\
& X \parallel Y \mid X \setminus a \mid P \\
(\text{Communication actions}) \quad & \alpha = a \mid \bar{a} \mid \tau \\
(\text{Prefixes}) \quad & \pi = \alpha \mid \sigma \\
(\text{Runtime prefixes}) \quad & \rho = \pi \mid \sigma_{\perp}
\end{aligned}$$

Figure 1: Syntax of revTPL

both the choice operator “+” and the discarded branch $b.Q$ disappear as an effect of the transition. In static approaches, the choice operator and the discarded branch are syntactically maintained, and process $a.P + b.Q$ can perform the transition below:

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

where $a[i].P + b.Q$ acts like P in forward transitions, 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 [LMM21, LP21].

2. THE REVERSIBLE TEMPORAL PROCESS LANGUAGE

In this section we define revTPL, an extension of Hennessy & Regan TPL (Temporal Process Language) [HR95] with causal-consistent reversibility following the static approach in the style of [PU07].

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

Processes (P, Q, \dots) describe timed interactions following [HR95]. We let \mathcal{A} be the set of action names a , $\bar{\mathcal{A}}$ the set of action conames \bar{a} . We use α to range over a , \bar{a} and internal actions τ . We assume $\bar{\bar{a}} = a$. In process $\pi.P$, prefix π can be a communication action α or a time action σ , and P is the continuation. Timeout $[P](Q)$ 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 process from CCS. For each recursive call A we assume a recursive definition $A \stackrel{def}{=} P$. We also assume recursion to be guarded, hence recursive variables can only occur under prefix.

Configurations (X, Y, \dots) 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, \dots) . Configurations are processes with (possibly) some computational history (i.e., prefixes marked with keys): $\pi[i].X$ is the configuration that has already executed π , and the execution of such π is identified by key i . Configuration $[X][\underline{i}](Y)$ is executing the main branch X whereas $[X][\underline{j}](Y)$ is executing Y . Some TPL processes, namely patient processes like $\alpha.P$ illustrated earlier in (1.4), allow time to pass without changing their own structure. This is an issue in revTPL, since it may lead different parallel components to have a different understanding of the

passage of time, while we want time to pass at the same pace for each parallel component. For this reason, to record that time has passed for a patient process, we use a special prefix $\sigma_{\perp}[i].X$. Namely, $\sigma_{\perp}[i].X$ is the configuration which has patiently registered the passage of time along with key i . Prefix $\sigma_{\perp}[i]$ differs from $\sigma[i]$ since the former is only for the current execution (patient delays *may* happen but do not always have to), while the latter requires time to pass in each possible execution (see Example 2.4). We will discuss this issue in more detail in Section 4. We use ρ to denote either π or σ_{\perp} .

A configuration can be thought of as a context with actions that have already been executed, each associated to a key, containing a process 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 time action traced by different processes, e.g., by two parallel delays. A configuration P can be thought of as the initial state of a computation, where no action has been executed yet. We call such configurations *standard*. Definition 2.1 formalises this notion via function $\mathbf{keys}(X)$ that returns the set of keys of a given configuration.

Definition 2.1 (Standard configuration). The set of keys of a configuration X , written $\mathbf{keys}(X)$, is inductively defined as follows:

$$\begin{aligned} \mathbf{keys}(P) &= \emptyset & \mathbf{keys}(\rho[i].X) &= \{i\} \cup \mathbf{keys}(X) & \mathbf{keys}(X \setminus a) &= \mathbf{keys}(X) \\ \mathbf{keys}([Y][\vec{i}](X)) &= \mathbf{keys}([X][\overset{\leftarrow}{i}](Y)) & &= \{i\} \cup \mathbf{keys}(X) \\ \mathbf{keys}(X + Y) &= \mathbf{keys}(X \parallel Y) & &= \mathbf{keys}(X) \cup \mathbf{keys}(Y) \end{aligned}$$

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

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

Definition 2.2 (Not-acted configuration). The not-acted predicate $\mathbf{nact}(\cdot)$ is inductively defined as:

$$\begin{aligned} \mathbf{nact}(\mathbf{0}) &= \mathbf{nact}(A) = \mathbf{nact}([X](Y)) = \mathbf{nact}(\pi.X) = \mathbf{tt} \\ \mathbf{nact}(\alpha[i].X) &= \mathbf{nact}([X][\overset{\leftarrow}{i}](Y)) = \mathbf{ff} \\ \mathbf{nact}(\sigma[i].X) &= \mathbf{nact}(\sigma_{\perp}[i].X) = \mathbf{nact}(X \setminus a) = \mathbf{nact}([Y][\vec{i}](X)) = \mathbf{nact}(X) \\ \mathbf{nact}(X \parallel Y) &= \mathbf{nact}(X + Y) = \mathbf{nact}(X) \wedge \mathbf{nact}(Y) \end{aligned}$$

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

Basic standard configurations are always not-acted (first line of Definition 2.2). Indeed, it is not possible to reach a configuration $\pi.X$ where X is acted. In the second line, a configuration that has executed communication actions is acted. In particular, we will see that $[X][\overset{\leftarrow}{i}](Y)$ is only reachable via a communication action. The configurations in the third line are not-acted if their continuations are not-acted. For parallel composition and choice, $\mathbf{nact}(\cdot)$ is defined as a conjunction. For example $\mathbf{nact}(\alpha[i].P \parallel \beta.Q) = \mathbf{ff}$ and $\mathbf{nact}(\alpha[i].P + \beta.Q) = \mathbf{ff}$. Note that in a choice configuration $X + Y$, at most one between X and Y can be acted. Whereas $\mathbf{std}(X)$ implies $\mathbf{nact}(X)$, the opposite implication does not hold. For example, $\mathbf{std}(\sigma[i].\mathbf{0}) = \mathbf{ff}$ but $\mathbf{nact}(\sigma[i].\mathbf{0}) = \mathbf{tt}$.

$$\begin{array}{c}
\text{PACT } \alpha.P \xrightarrow{\sigma[i]} \sigma_{\perp}[i].\alpha.P \qquad \text{RACT } \pi.P \xrightarrow{\pi[i]} \pi[i].P \qquad \text{IDLE } \mathbf{0} \xrightarrow{\sigma[i]} \sigma_{\perp}[i].\mathbf{0} \\
\\
\text{ACT } \frac{X \xrightarrow{\pi[j]} X' \quad j \neq i}{\rho[i].X \xrightarrow{\pi[j]} \rho[i].X'} \qquad \text{STOUT } \frac{X \xrightarrow{\tau} \text{std}(X) \quad \text{std}(Y)}{[X](Y) \xrightarrow{\sigma[i]} [X][\dot{i}](Y)} \\
\\
\text{SWAIT } \frac{Y \xrightarrow{\pi[j]} Y' \quad j \neq i}{[X][\dot{i}](Y) \xrightarrow{\pi[j]} [X][\dot{i}](Y')} \qquad \text{TOUT } \frac{X \xrightarrow{\alpha[i]} X' \quad \text{std}(Y)}{[X](Y) \xrightarrow{\alpha[i]} [X'][\dot{i}](Y)} \\
\\
\text{WAIT } \frac{X \xrightarrow{\pi[j]} X' \quad j \neq i}{[X][\dot{i}](Y) \xrightarrow{\pi[j]} [X'][\dot{i}](Y)} \qquad \text{SYNW } \frac{X \xrightarrow{\sigma[i]} X' \quad Y \xrightarrow{\sigma[i]} Y' \quad (X \parallel Y) \xrightarrow{\tau}}{X \parallel Y \xrightarrow{\sigma[i]} X' \parallel Y'} \\
\\
\text{PAR } \frac{X \xrightarrow{\alpha[i]} X' \quad i \notin \text{keys}(Y)}{X \parallel Y \xrightarrow{\alpha[i]} X' \parallel Y} \qquad \text{SYN } \frac{X \xrightarrow{\alpha[i]} X' \quad Y \xrightarrow{\bar{\alpha}[i]} Y'}{X \parallel Y \xrightarrow{\tau[i]} X' \parallel Y'} \\
\\
\text{CHOW } \frac{X \xrightarrow{\sigma[i]} X' \quad Y \xrightarrow{\sigma[i]} Y'}{X + Y \xrightarrow{\sigma[i]} X' + Y'} \qquad \text{CHO } \frac{X \xrightarrow{\alpha[i]} X' \quad \text{nact}(Y) \quad i \notin \text{keys}(Y)}{X + Y \xrightarrow{\alpha[i]} X' + Y} \\
\\
\text{HIDE } \frac{X \xrightarrow{\pi[i]} X' \quad \pi \notin \{a, \bar{a}\}}{X \setminus a \xrightarrow{\pi[i]} X' \setminus a} \qquad \text{CONST } \frac{A \stackrel{\text{def}}{=} P \quad P \xrightarrow{\pi[i]} X}{A \xrightarrow{\pi[i]} X}
\end{array}$$

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

Figure 2: revTPL forward LTS

2.2. Semantics of revTPL. We denote with \mathcal{A}^t the set $\mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau, \sigma\}$ of actions and let π to range over the set \mathcal{A}^t . We define the set of all the labels $\mathcal{L} = \mathcal{A}^t \times \mathcal{K}$. The labels associate each $\pi \in \mathcal{A}^t$ to a key i . 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.

Definition 2.3 (Semantics). The operational semantics of revTPL is given by two Labelled Transition Systems (LTSs) defined on the same set of all configurations \mathcal{X} , and the set of all labels \mathcal{L} : a forward LTS $(\mathcal{X}, \mathcal{L}, \rightarrow)$ and a backward LTS $(\mathcal{X}, \mathcal{L}, \leftarrow)$. We define $\mapsto = \rightarrow \cup \leftarrow$, where \rightarrow and \leftarrow 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 reflexive and transitive closure. We use notation $X \xrightarrow{\tau} X'$ (resp. $X \xrightarrow{\tau} X'$) when there are no configuration X' and key i such that $X \xrightarrow{\tau[i]} X'$ (resp. $X \xrightarrow{\tau[i]} X'$).

We now discuss the rules of the forward semantics (Figure 2). Rule [PACT] describes patient actions: in TPL process $\alpha.P$ can make a time step to itself. This kind of actions allows a process to wait indefinitely until it can communicate (by patience [HR95]). However, in revTPL we need to track passage of time, hence rule [PACT] adds a $\sigma_{\perp}[i]$ prefix in front of the configuration, with a key i . Rule [RACT] executes actions $\alpha[i]$ or $\sigma[i]$ on a prefix process. Observe that, unlike patient time actions on $\alpha.P$, a time action on $\sigma.P$ corresponds to a deliberate and planned time consuming action and, therefore, it executes the σ prefix, hence no σ_{\perp} prefix needs to be added. Rule [IDLE] registers passage of time on a $\mathbf{0}$ configuration by adding a $\sigma_{\perp}[i]$ prefix to it. Rule [ACT] lifts actions of the continuation X on configurations where prefix $\rho[i]$ has already been executed. Side condition $j \neq i$ ensures freshness of j is preserved. Rules [STOUT] and [SWAIT] model timeouts. In rule [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 configuration X is stuck. The negative premise on [STOUT] can be encoded into a decidable positive one as shown in Appendix B. In rule [TOUT] instead the main configuration can execute and the timeout does not trigger. Rule [SWAIT] (resp. [WAIT]) models transitions inside a timeout configuration where the Y (resp. X) branch has been previously taken. The semantics of timeout construct becomes clearer in the larger context of parallel configurations, when looking at rule [SYNW]. Rule [SYNW] models time passing for parallel configurations. The negative premise ensures that, in case X or Y is a timeout configuration, timeout can trigger only if no synchronisation may occur, that is if the configurations are stuck. [SYNW] requires time to pass in the same way (an action σ is taken by both components, with the same key i) for the whole system. Rules [PAR] (and symmetric) and [SYN] are as usual for communication actions and allow parallel configurations to either proceed independently or to synchronise. In the latter case, the keys need to coincide. Defining the semantics of choice configuration $X + Y$ requires special care to ensure time-determinism (recall, choices are only decided via communication actions). Also, we need to record time actions to be able to reverse them correctly (cfr. Loop Lemma, discussed later on in Lemma 4.1). Rule [CHOW] describes the passage of time along a choice configuration $X + Y$. Since time does not decide a choice, both branches have to execute the same time action with the same key. 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., $\mathbf{nact}(Y)$). This ensures that choices can be decided by a communication action only. Let us note that even in case of a decided choice, that is a choice configuration in which one of the two branches has performed a communication action, time actions are registered by both configurations. For example, the configuration $a.\mathbf{0} + \sigma.\mathbf{0}$ can execute the following transitions:

$$a.\mathbf{0} + \sigma.\mathbf{0} \xrightarrow{a[i]} a[i].\mathbf{0} + \sigma.\mathbf{0} \xrightarrow{\sigma[j]} a[i].\sigma_{\perp}[j].\mathbf{0} + \sigma[j].\mathbf{0}$$

After the $a[i]$ action, even if the left branch of the choice has been selected, both branches participate to the time action $\sigma[j]$. Rules [HIDE] and [CONST] are standard.

The rules of the backward semantics, in Figure 3, undo communication and time actions executed under the forward semantics. Backward rules are symmetric to the forward ones.

Now that we have introduced both the forward and the backward rules we can clarify the difference between σ_{\perp} and σ .

Example 2.4. Let us consider the patient process $a.P$. We can have the following derivation:

$$a.P \xrightarrow{\sigma[i]} \sigma_{\perp}[i].a.P \xleftarrow{\sigma[i]} a.P \xrightarrow{a[j]} a[j].P \quad (2.1)$$

$$\begin{array}{c}
\text{PACT } \sigma_{\perp}[i].\alpha.P \xrightarrow{\sigma[i]} \alpha.P \qquad \text{RACT } \pi[i].P \xrightarrow{\pi[i]} \pi.P \qquad \text{IDLE } \sigma_{\perp}[i].\mathbf{0} \xrightarrow{\sigma[i]} \mathbf{0} \\
\\
\text{ACT } \frac{X \xrightarrow{\pi[j]} X' \quad j \neq i}{\rho[i].X \xrightarrow{\pi[j]} \rho[i].X'} \qquad \text{STOUT } \frac{X \xrightarrow{\tau} \text{std}(X) \quad \text{std}(Y)}{[X][\dot{i}](Y) \xrightarrow{\sigma[i]} [X](Y)} \\
\\
\text{SWAIT } \frac{Y \xrightarrow{\pi[j]} Y' \quad j \neq i}{[X][\dot{i}](Y) \xrightarrow{\pi[j]} [X][\dot{i}](Y')} \qquad \text{TOUT } \frac{X \xrightarrow{\alpha[i]} X' \quad \text{std}(Y)}{[X][\dot{i}](Y) \xrightarrow{\alpha[i]} [X'](Y)} \\
\\
\text{WAIT } \frac{X \xrightarrow{\pi[j]} X' \quad j \neq i}{[X][\dot{i}](Y) \xrightarrow{\pi[j]} [X'][\dot{i}](Y)} \qquad \text{SYNW } \frac{X \xrightarrow{\sigma[i]} X' \quad Y \xrightarrow{\sigma[i]} Y' \quad (X \parallel Y) \xrightarrow{\tau}}{X \parallel Y \xrightarrow{\sigma[i]} X' \parallel Y'} \\
\\
\text{PAR } \frac{X \xrightarrow{\alpha[i]} X' \quad i \notin \text{keys}(Y)}{X \parallel Y \xrightarrow{\alpha[i]} X' \parallel Y} \qquad \text{SYN } \frac{X \xrightarrow{\alpha[i]} X' \quad Y \xrightarrow{\bar{\alpha}[i]} Y'}{X \parallel Y \xrightarrow{\tau[i]} X' \parallel Y'} \\
\\
\text{CHOW } \frac{X \xrightarrow{\sigma[i]} X' \quad Y \xrightarrow{\sigma[i]} Y'}{X + Y \xrightarrow{\sigma[i]} X' + Y'} \qquad \text{CHO } \frac{X \xrightarrow{\alpha[i]} X' \quad \text{nact}(Y) \quad i \notin \text{keys}(Y)}{X + Y \xrightarrow{\alpha[i]} X' + Y} \\
\\
\text{HIDE } \frac{X \xrightarrow{\pi[i]} X' \quad \pi \notin \{a, \bar{a}\}}{X \setminus a \xrightarrow{\pi[i]} X' \setminus a} \qquad \text{CONST } \frac{A \stackrel{\text{def}}{=} P \quad X \xrightarrow{\pi[i]} P}{X \xrightarrow{\pi[i]} A}
\end{array}$$

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

Figure 3: revTPL backward LTS

where $a.P$ executes forwards in two different ways: first by letting time pass, later on by interacting on a . Notice that for these interactions to be possible in a larger context we need the context to have changed as well.

We can try to have a similar derivation using process $\sigma.a.P$ instead, but the final outcome is not the same:

$$\sigma.a.P \xrightarrow{\sigma[i]} \sigma[i].a.P \xrightarrow{\sigma[i]} \sigma.a.P \quad (2.2)$$

Indeed, at this stage $\sigma.a.P$ cannot interact on a . In general, σ requires time to pass in every possible computation, while σ_{\perp} does not. \diamond

Definition 2.5 (Reachable configurations). A configuration X is reachable if there exist a process P and a derivation $P \mapsto^* X$.

Basically, a configuration is reachable if it can be obtained via forward and backward actions from a standard configuration.

3. 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 [PU07]) 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 configuration.

Definition 3.1 (History forgetting map). The *history forgetting map* $\phi^h : \mathcal{X} \rightarrow \mathcal{P}$ is inductively defined as follows:

$$\begin{aligned} \phi^h(P) &= P & \phi^h(\rho[i].X) &= \phi^h(X) \\ \phi^h(\lfloor X \rfloor [\overset{\leftarrow}{i}](Y)) &= \phi^h(X) & \phi^h(\lfloor X \rfloor [\vec{i}](Y)) &= \phi^h(Y) \\ \phi^h(X \parallel Y) &= \phi^h(X) \parallel \phi^h(Y) & \phi^h(X \setminus a) &= \phi^h(X) \setminus a \\ \phi^h(X + Y) &= \begin{cases} \phi^h(X) & \text{if } \neg \mathbf{nact}(X) \wedge \mathbf{nact}(Y) \\ \phi^h(Y) & \text{if } \neg \mathbf{nact}(Y) \wedge \mathbf{nact}(X) \\ \phi^h(X) + \phi^h(Y) & \text{otherwise} \end{cases} \end{aligned}$$

The definition above deletes all the information about history from a configuration X , hence it is the identity on standard configuration P . Even more, each configuration is mapped into a standard one. Notice that in a non-standard timeout, only the chosen branch is taken. In TPL time cannot decide choices. This is reflected into the definition of $\phi^h(X + Y)$, where a branch disappears only if the other one did at least a communication action.

Notably, the restriction of ϕ^h to untimed configurations (namely configurations containing neither timeouts nor σ prefixes nor σ_{\perp} prefixes) is a map from CCSK [PU07] to CCS. Following the notation of Appendix A, we will indicate with $\rightarrow_{\mathfrak{t}}$ the semantics of TPL [HR95], reported in Appendix A.3, and with $\mapsto_{\mathfrak{k}}$ the semantics of CCSK [PU07], reported in Appendix A.2.

Proposition 3.2 (Embedding of TPL). *Let X be a reachable revTPL configuration:*

- (1) if $X \xrightarrow{\pi[i]} Y$ then $\phi^h(X) \xrightarrow{\pi}_{\mathfrak{t}} \phi^h(Y)$;
- (2) if $\phi^h(X) \xrightarrow{\pi}_{\mathfrak{t}} Q$ then for any $i \in \mathcal{K} \setminus \mathbf{keys}(X)$ there is Y such that $X \xrightarrow{\pi[i]} Y$ with $\phi^h(Y) = Q$.

Proof. (1) : by induction on the derivation $X \xrightarrow{\pi[i]} Y$, with a case analysis on the last applied rule. We detail a few sample rules.

If the move is by rule [PACT] then we have $Y = \sigma_{\perp}[i].\alpha.X_1$, with $\phi^h(X) = \phi^h(Y)$, and in TPL we have a corresponding state-preserving move with label σ derived using rule ACT₂ in Figure 12.

In the case of rule [ACT], $X = \rho[i].Z$ and $Y = \rho[i].Z'$ with $Z \xrightarrow{\pi[j]} Z'$. By inductive hypothesis, in TPL $\phi^h(Z) \xrightarrow{\pi}_{\mathfrak{t}} \phi^h(Z')$. Since $\phi^h(X) = \phi^h(Z)$ and $\phi^h(Y) = \phi^h(Z')$ we are done. The cases for [CONST] and [HIDE] are similar by induction. The cases for [SYNW] and [CHOW] follow by induction as well.

If the last applied rule is [CHO], then we have that $X = X_1 + X_2$ with $X_1 \xrightarrow{\alpha[i]} X'_1$ and $\mathbf{nact}(X_2)$. Also, $Y = X'_1 + X_2$ with X'_1 acted. Hence, $\phi^h(Y) = \phi^h(X'_1 + X_2) = \phi^h(X'_1)$. We consider the case $\mathbf{nact}(X_1)$, the other one is simpler. By definition, we have that $\phi^h(X_1 + X_2) = \phi^h(X_1) + \phi^h(X_2)$. Since $X_1 \xrightarrow{\alpha[i]} X'_1$, by applying the inductive

hypothesis we also have that $\phi^h(X_1) \xrightarrow{\alpha}_t \phi^h(X'_1)$. Also, since the label is not a σ action, in TPL we can use rule SUM_1 in Figure 12 that from $\phi^h(X_1) \xrightarrow{\alpha}_t \phi^h(X'_1)$ allows one to derive $\phi^h(X_1) + \phi^h(X_2) \xrightarrow{\alpha}_t \phi^h(X'_1)$, as desired.

If the last applied rule is $[\text{STOUT}]$ then $\lfloor X_1 \rfloor(X_2) \xrightarrow{\sigma[i]} \lfloor X_1 \rfloor[\underline{i}](X_2)$ with X_1 and X_2 standard, and X_1 that can not perform τ steps. By inductive hypothesis $\phi^h(X_1)$ can not perform τ steps in TPL, hence in TPL we can use rule THEN_2 in Figure 12 to derive $\phi^h(\lfloor X_1 \rfloor(X_2)) = \lfloor X_1 \rfloor(X_2) \xrightarrow{\sigma}_t X_2 = \phi^h(\lfloor X_1 \rfloor[\underline{i}](X_2))$ as desired.

(2) : by induction on the definition of $\phi^h(X)$ (structural induction on X). Let us first assume X standard, hence $\phi^h(X) = X$. Let us consider $X = \alpha.P$. In TPL, $\alpha.P$ can make a state preserving transition σ and the corresponding revTPL configuration can match it: $X \xrightarrow{\sigma[i]} \sigma_\perp[i].X$, with $\phi^h(\sigma_\perp[i].X) = X$. Alternatively, in TPL, $\alpha.P \xrightarrow{\alpha}_t P$. The thesis follows since $\alpha.P \xrightarrow{\alpha[i]} \alpha[i].P$ with $\phi^h(\alpha[i].P) = P$. The other cases are similar, but using the induction hypothesis.

Let us now assume X non standard. The most interesting case is when $X = X_1 + X_2$. Let us consider X_1 acted (the case where X_2 is acted is symmetric). In this case $\phi^h(X_1 + X_2) = \phi^h(X_1)$ hence the thesis follows by inductive hypothesis using rule $[\text{CHOW}]$ for σ actions and $[\text{CHO}]$ for communication actions. If both X_1 and X_2 are not acted, then $\phi^h(X_1 + X_2) = \phi^h(X_1) + \phi^h(X_2)$. We now have two cases, either $\pi = \sigma$ or $\pi = \alpha$. If $\pi = \sigma$ we have that by rule SUM_3 in Figure 12 $\phi^h(X_1) \xrightarrow{\sigma}_t Z_1$ and $\phi^h(X_2) \xrightarrow{\sigma}_t Z_2$ allow one to derive $\phi^h(X_1) + \phi^h(X_2) \xrightarrow{\sigma}_t Z_1 + Z_2$. By inductive hypotheses we have that there exist X'_1 and X'_2 such that $X_1 \xrightarrow{\sigma[i]} X'_1$ and $X_2 \xrightarrow{\sigma[i]} X'_2$ with $\phi^h(X'_1) = Z_1$ and $\phi^h(X'_2) = Z_2$. We can then apply rule $[\text{CHOW}]$ to derive $X_1 + X_2 \xrightarrow{\sigma[i]} X'_1 + X'_2$. Since X'_1 and X'_2 are still not-acted we can conclude by noticing that $\phi^h(X'_1 + X'_2) = \phi^h(X'_1) + \phi^h(X'_2) = Z_1 + Z_2$. The case for $\pi = \alpha$ is similar. \square

We can describe the above correspondence between revTPL and TPL in a more abstract way by adapting the notion of (strong) timed bisimulation [LY97] to relate configurations from two calculi.

Definition 3.3 (Timed bisimulation). A binary relation \mathcal{R} on $\mathcal{X} \times \mathcal{P}$ is a strong timed bisimulation between revTPL and TPL if $(X, P) \in \mathcal{R}$ implies that

- (1) if $X \xrightarrow{\pi[i]} Y$, then there exists Q such that $P \xrightarrow{\pi}_t Q$ and $(Y, Q) \in \mathcal{R}$;
- (2) if $P \xrightarrow{\pi}_t Q$, then there exist Y and i such that $X \xrightarrow{\pi[i]} Y$ and $(Y, Q) \in \mathcal{R}$.

The largest strong timed bisimulation is called strong timed equivalence, denoted \sim .

We can now relate revTPL and TPL as follows:

Theorem 3.4. *For each reachable revTPL configuration X we have that $X \sim \phi^h(X)$.*

Proof. It is sufficient to show that the relation $\mathcal{R} = \{(X, P) \mid \phi^h(X) = P\}$ is a strong timed bisimulation. Let us check the conditions. If $X \xrightarrow{\pi[i]} Y$ then thanks to Proposition 3.2 we have that $\phi^h(X) \xrightarrow{\pi}_t \phi^h(Y)$ with $\phi^h(Y) = Q$, and we have that $(Y, Q) \in \mathcal{R}$. If $P \xrightarrow{\pi}_t Q$, thanks to Proposition 3.2 we have that $X \xrightarrow{\pi[i]} Y$ with $\phi^h(Y) = Q$, and we have that $(Y, Q) \in \mathcal{R}$, as desired. \square

Also, TPL is a conservative extension of CCS. This is stated in [HR95], albeit 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 [PU07]. 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 ϕ^{\dagger} . We denote with \mathcal{X}^k the set of untimed reversible configurations of revTPL, which coincides with the set of all CCSK configurations (which is defined in Appendix A.2). The set inclusion $\mathcal{X}^k \subset \mathcal{X}$ holds.

Definition 3.5 (Time forgetting map). The *time forgetting map* $\phi^{\dagger} : \mathcal{X} \rightarrow \mathcal{X}^k$ is inductively defined as follows:

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

Notably, the restriction of ϕ^{\dagger} to standard configurations is a map from TPL to CCS.

The most interesting aspect in the definition above is that the timeout operator $\lfloor X \rfloor(Y)$ is rendered as a sum. This also happens for the decorated configurations $\lfloor X \rfloor[\underline{i}](Y)$ and $\lfloor X \rfloor[\underline{i}](Y)$. We will further discuss this design decision after Proposition 3.6. Also, since we are relating a timed semantics with an untimed one (CCSK), the σ actions performed by the timed semantics are not reflected in CCSK.

Proposition 3.6 (Embedding of CCSK [PU07]). *Let X be a reachable revTPL configuration. We have:*

- (1) if $X \xrightarrow{\alpha[i]} Y$ then $\phi^{\dagger}(X) \xrightarrow{\alpha[i]}_{\mathbf{k}} \phi^{\dagger}(Y)$;
- (2) if $X \xrightarrow{\alpha[i]} Y$ then $\phi^{\dagger}(X) \xrightarrow{\alpha[i]}_{\mathbf{k}} \phi^{\dagger}(Y)$;
- (3) if $X \xrightarrow{\sigma[i]} Y$ then $\phi^{\dagger}(X) = \phi^{\dagger}(Y)$.

Proof. **(1)** : by induction on the derivation $X \xrightarrow{\alpha[i]} Y$, with a case analysis on the last applied rule. The proof goes along the lines of the proof of Proposition 3.2, using the rules reported in Figures 8 and 9 in Appendix A.2.

(2) : similar to the case above, using $X \xrightarrow{\alpha[i]} Y$ instead of $X \xrightarrow{\alpha[i]} Y$.

(3) : by induction on the derivation of $X \xrightarrow{\sigma[i]} Y$ (the case of backward transitions is analogous), with a case analysis on the last applied rule. Basic cases are (i) rules [PACT] and [IDLE], creating a σ_{\perp} , (ii) σ prefixes, and (iii) timeouts. In case (i) we have $Y = \sigma_{\perp}[i].X$, hence $\phi^{\dagger}(Y) = \phi^{\dagger}(X)$. In case (ii) we have $\sigma.P \xrightarrow{\sigma[i]} \sigma[i].P$, hence $\phi^{\dagger}(\sigma.P) = P = \phi^{\dagger}(\sigma[i].P)$. In case (iii) we have $\lfloor X_1 \rfloor(X_2) \xrightarrow{\sigma[i]} \lfloor X_1 \rfloor[\underline{i}](X_2)$ with $\phi^{\dagger}(\lfloor X_1 \rfloor(X_2)) = \phi^{\dagger}(X_1) + \phi^{\dagger}(X_2) = \phi^{\dagger}(\lfloor X_1 \rfloor[\underline{i}](X_2))$ as desired. Inductive cases follow by inductive hypothesis. \square

Notably, it is not always the case that transitions of the underlying untimed configuration can be matched in a timed setting. Think, e.g., of the Erlang program in Example 0.1 (and its formalisation in Section 1.1), for a counterexample. Indeed, in the example, the

communication between the two processes A and B is allowed in the underlying untimed model, but ruled out by incompatible timing constraints. Also, let us consider the simple process $X = [a](b)$ and its untimed version $\phi^{\mathfrak{t}}(X) = a + b$. The untimed process can execute the right branch as follows

$$\phi^{\mathfrak{t}}(X) \xrightarrow{b[i]} a + b[i]$$

To match this action, X has first to perform a time action and only afterwards it can take the b action, as follows:

$$X \xrightarrow{\sigma[j]} [a][\underline{j}](b) \xrightarrow{b[i]} [a][\underline{j}](b[i])$$

Moreover, there are also cases where actions cannot be matched, not even after time actions. Indeed, the timeout operator $[P](Q)$ acts as a choice with left priority. For example, let us consider the process $X = [(a \parallel \bar{a})](b)$. We have that $\phi^{\mathfrak{t}}(X)$ can perform the b action as follows

$$\phi^{\mathfrak{t}}(X) = (a \parallel \bar{a}) + b \xrightarrow{b[i]} (a \parallel \bar{a}) + b[i]$$

but this action can never be matched by X , as **revTPL** maximal progress forces the internal synchronisation over time passage. Hence we can only apply rule **STOUT** in Figure 2:

$$[(a \parallel \bar{a})](b) \xrightarrow{\tau[i]} [a[i] \parallel \bar{a}[i]][\underline{j}](b)$$

and the resulting configuration cannot execute b .

Due to the examples above, we cannot characterise the relation between X and $\phi^{\mathfrak{t}}(X)$ as a bisimulation, as for $\phi^{\mathfrak{h}}$, but we can only prove that $\phi^{\mathfrak{t}}(X)$ simulates X . As before, we need to modify notions of simulation for reversible configurations from the literature (e.g., [NMV90, LP21]) to relate configurations from two calculi, and to keep time into account.

Definition 3.7 (Back and forward simulation). A binary relation \mathcal{R} on $\mathcal{X} \times \mathcal{X}^k$ is a back and forward simulation if $(X, R) \in \mathcal{R}$ implies that

- (1) if $X \xrightarrow{\alpha[i]} Y$, then there exists S such that $R \xrightarrow{\alpha[i]}_{\mathbf{k}} S$ and $(Y, S) \in \mathcal{R}$;
- (2) if $X \xrightarrow{\alpha[i]} Y$, then there exists S such that $R \xrightarrow{\alpha[i]}_{\mathbf{k}} S$ and $(Y, S) \in \mathcal{R}$;
- (3) if $X \xrightarrow{\sigma[i]} Y$, then $(Y, R) \in \mathcal{R}$.

The largest back and forward simulation is denoted by \lesssim .

Theorem 3.8. *For each reachable **revTPL** configuration X we have that $X \lesssim \phi^{\mathfrak{t}}(X)$.*

Proof. It is sufficient to show that the relation $\mathcal{R} = \{(X, R) \mid \phi^{\mathfrak{t}}(X) = R\}$ is a back and forward simulation. It is easy to check the conditions of Definition 3.7 using Proposition 3.6. \square

Figure 4 summarises our results: if we remove the timed behaviour from a **revTPL** configuration we get a CCSK term, with the same behaviour apart for timed aspects, thanks to Proposition 3.6. On the other side, if from **revTPL** we remove history information we get a TPL term (matching its forward behaviour thanks to Proposition 3.2). 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 [PU07], showing that by removing reversibility and history information from CCSK we get CCS. Notably the two forgetting maps commute.

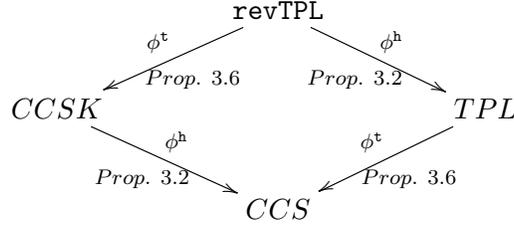


Figure 4: Forgetting maps.

Proposition 3.9. *For each reachable revTPL configuration X we have $\phi^h(\phi^t(X)) = \phi^t(\phi^h(X))$.*

Proof. By structural induction on X . □

4. REVERSIBILITY IN revTPL

In a fully reversible calculus any computation can be undone. This is a fundamental property of reversibility [DK04, LPU20], called the Loop Lemma, and revTPL enjoys it. Formally:

Lemma 4.1 (Loop Lemma). *If X is a reachable revTPL configuration, then $X \xrightarrow{\pi[i]} X' \iff X' \xrightarrow{\pi[i]} X$*

Proof. We have two directions. The forward one trivially holds, since for each forward rule of Figure 2 there exists a symmetric one backwards in Figure 3. The backward case requires more attention, and we proceed by induction on $X \xrightarrow{\pi[i]} Y$, with a case analysis on the last applied rule. We can further distinguish the cases according to whether $\pi = \sigma$, $\pi = \tau$ or $\pi = \alpha$. We will just consider one instance of each case, the others are similar.

$\pi = \sigma$: using rule [CHOW] we have that $X = X_1 + X_2$, $X_1 \xrightarrow{\sigma[i]} X'_1$, $X_2 \xrightarrow{\sigma[i]} X'_2$. Since by reachability of X we have the reachability of X_1 and X_2 , by inductive hypothesis we have that $X'_1 \xrightarrow{\sigma[i]} X_1$ and $X'_2 \xrightarrow{\sigma[i]} X_2$, so we can apply the forward version of rule [CHOW], as desired.

$\pi = \tau$: using rule [SYN] by hypothesis we have that $X = X_1 \parallel X_2$ with $X_1 \xrightarrow{\alpha[i]} X'_1$ and $X_2 \xrightarrow{\bar{\alpha}[i]} X'_2$. By applying the inductive hypothesis we get $X'_1 \xrightarrow{\alpha[i]} X_1$ and $X'_2 \xrightarrow{\bar{\alpha}[i]} X_2$, and we can derive $X'_1 \parallel X'_2 \xrightarrow{\tau[i]} X_1 \parallel X_2$, as desired.

$\pi = \alpha$: using rule [CHO] by hypothesis we have that $X = X_1 + X_2$, $X_1 \xrightarrow{\alpha[i]} X'_1$ and $\mathbf{nact}(X_2)$. By applying the inductive hypothesis we have that $X'_1 \xrightarrow{\alpha[i]} X_1$ and we can derive $X'_1 + X_2 \xrightarrow{\alpha[i]} X_1 + X_2$ as desired. □

Another fundamental property of causal-consistent reversibility is the so-called causal-consistency [DK04, LPU20], which essentially states that we store the correct amount of causal information. In order to discuss it, we now borrow some definitions from [DK04]. We use t, t', s, s' to range over transitions. In a transition $t : X \xrightarrow{\pi[i]} Y$ we call X the *source* of

the transition, and Y the *target* of the transition. Two transitions are said to be *cointial* if they have the same source, and *cofinal* if they have the same target. Given a transition t , we indicate with \underline{t} its reverse, that is if $t : X \xrightarrow{\pi[i]} Y$ (resp., $t : X \xleftarrow{\pi[i]} Y$) then $\underline{t} : Y \xleftarrow{\pi[i]} X$ (resp., $\underline{t} : Y \xrightarrow{\pi[i]} X$). The notions of source, target, cointiality, and cofinality naturally extend to paths. We let χ, ω to range over sequences of transitions, which we call *paths*, and with ϵ_X we indicate the empty sequence starting and ending at X . We denote as $|\chi|$ the number of transitions in path χ . Moreover, we indicate with $\chi_1\chi_2$ the composition of the two paths χ_1 and χ_2 when they are composable, that is when the target of χ_1 coincides with the source of χ_2 .

Definition 4.2 (Causal Equivalence). Let \simeq be the smallest equivalence on paths closed under composition and satisfying:

- (1) if $t : X \xrightarrow{\pi_1[i]} Y_1$ and $s : X \xrightarrow{\pi_2[j]} Y_2$ are independent, and $s' : Y_1 \xrightarrow{\pi_2[j]} Z$, $t' : Y_2 \xrightarrow{\pi_1[i]} Z$ then $ts' \simeq st'$;
- (2) $t\underline{t} \simeq \epsilon$ and $\underline{t}t \simeq \epsilon$

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 4.3 (Causal Consistency (CC)). An LTS is causal consistent if for any cointial and cofinal paths χ and ω we have $\chi \simeq \omega$.

Intuitively, if cointial 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.

4.1. Independence. We now define a notion of independence between **revTPL** cointial transitions, based on a causality preorder (inspired by [LP21]) on keys. Intuitively, independent transitions can be executed in any order (we will formalise this as Property 4.11), while transitions which are not independent represent a choice: either one is executed, or the other.

Definition 4.4 (Partial order on keys). The function $\text{po}(\cdot) : \mathcal{X} \mapsto 2^{(\mathcal{K} \times \mathcal{K})}$ is inductively defined below. It takes a configuration $X \in \mathcal{X}$ and computes a set of ordered pairs of keys which is the set of causal relations among the keys in X .

$$\begin{aligned} \text{po}(P) &= \emptyset & \text{po}(X \setminus a) &= \text{po}(X) \\ \text{po}(X \parallel Y) &= \text{po}(X + Y) = \text{po}(\lfloor X \rfloor(Y)) = \text{po}(X) \cup \text{po}(Y) \\ \text{po}(\rho[i].X) &= \text{po}(\lfloor X \rfloor[\underline{i}](Y)) = \text{po}(\lfloor Y \rfloor[\underline{i}](X)) = \{i < j \mid j \in \text{keys}(X)\} \cup \text{po}(X) \end{aligned}$$

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

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. In particular, $i < j$ means that key i causes key j . This takes into account both structural causality given by the structure of a configuration (e.g., a prefix causes its continuation) and causality raising from synchronisation and time, since synchronising actions and time actions corresponding to the same point in time have the same key.

Example 4.5. Let us compute the partial order on keys in

$$[a][\underline{i}](b[j].P) \parallel \sigma_{\perp}[i].c[k].d[w].Q \parallel \sigma_{\perp}[i].\bar{c}[k].R$$

We have:

$$\begin{aligned} \text{po}([a][\underline{i}](b[j].P)) &= \{i < j\} \cup \text{po}(P) = \{i < j\} \cup \emptyset = \{i < j\} \\ \text{po}(\sigma_{\perp}[i].c[k].d[w].Q) &= \{i < k\} \cup \{i < w\} \cup \text{po}(c[k].d[w].Q) \\ &= \{i < k\} \cup \{i < w\} \cup \{k < w\} \cup \text{po}(d[w].Q) \\ &= \{i < k\} \cup \{i < w\} \cup \{k < w\} \cup \emptyset \cup \text{po}(Q) \\ &= \{i < k\} \cup \{i < w\} \cup \{k < w\} \\ \text{po}(\sigma_{\perp}[i].\bar{c}[k].R) &= \{i < k\} \cup \text{po}(R) = \{i < k\} \end{aligned}$$

and hence, looking at the parallel composition:

$$\begin{aligned} \text{po}([a][\underline{i}](b[j].P) \parallel \sigma_{\perp}[i].c[k].d[w].Q \parallel \sigma_{\perp}[i].\bar{c}[k].R) = \\ \{i < j\} \cup \{i < k\} \cup \{i < w\} \cup \{k < w\} \cup \{i < k\} = \{i < j, i < k, i < w, k < w\} \quad \diamond \end{aligned}$$

We also need to understand whether two forward communication transitions are in conflict since either they involve a same prefix or they involve different branches of a choice.

Definition 4.6 (Forward communication conflict). Two forward communication transitions $t_1 : X \xrightarrow{\alpha_1[i]} Y$ and $t_2 : X \xrightarrow{\alpha_2[j]} Z$ with $i \neq j$ are in forward communication conflict iff the $\text{fcc}(Y, Z)$ predicate defined below holds:

$$\begin{aligned} \text{fcc}(\alpha[i].P, \alpha[j].P) &= \text{True} \\ \text{fcc}(P, P) &= \text{False} \\ \text{fcc}(Y_1 \parallel Y_2, Z_1 \parallel Z_2) &= \text{fcc}(Y_1, Z_2) \vee \text{fcc}(Y_2, Z_1) \\ \text{fcc}(Y_1 + Y_2, Z_1 + Z_2) &= (Y_1 \neq Z_1 \wedge Y_2 \neq Z_2) \vee \text{fcc}(Y_1, Z_1) \vee \text{fcc}(Y_2, Z_2) \\ \text{fcc}(Y_1 \setminus a, Z_1 \setminus a) &= \text{fcc}(Y_1, Z_1) \\ \text{fcc}(\rho[i].Y_1, \rho[i].Z_1) &= \text{fcc}(Y_1, Z_1) \\ \text{fcc}([Y_1][\underline{j}](Y_2), [Z_1][\underline{j}](Z_2)) &= \text{fcc}(Y_1, Z_1) \\ \text{fcc}([Y_1][\underline{i}](Y_2), [Z_1][\underline{i}](Z_2)) &= \text{fcc}(Y_1, Z_1) \\ \text{fcc}([Y_1][\underline{i}](Y_2), [Z_1][\underline{j}](Z_2)) &= \text{fcc}(Y_2, Z_2) \end{aligned}$$

For simplicity, the fcc predicate above is defined only for pairs of configurations which may arise from the same configuration. Notice that in the clause for choice, the only way for the two branches to be pairwise different, is that Y has chosen one of them, and Z the other. In this case the two actions are in conflict. However, in this case fcc may not be defined on the components. To avoid this issue, we consider the \vee operator to be a short circuit operator.

Example 4.7. Let us consider the configuration $X_1 = a[i].b.P$ and the two transitions

- $t_1 : X_1 \xrightarrow{b[j]} a[i].b[j].P = Y_1$ and
- $t_2 : X_1 \xrightarrow{b[z]} a[i].b[z].P = Z_1$.

We have that

$$\text{fcc}(a[i].b[j].P, a[i].b[z].P) = \text{fcc}(b[j].P, b[z].P) = \text{True}$$

Let us consider the configuration $X_2 = [a.\mathbf{0}][\underline{i}](b[j].(a.P + b.Q))$ and the two transitions

- $t_3 : X_2 \xrightarrow{a[z]} [a.\mathbf{0}][\underline{i}](b[j].(a[z].P + b.Q)) = Y_2$ and

$$\bullet t_4 : X_2 \xrightarrow{a[w]} [a.\mathbf{0}][\underline{i}](b[j].(a.P + b[w].Q)) = Z_2.$$

We have that

$$\begin{aligned} & \text{fcc}([a.\mathbf{0}][\underline{i}](b[j].(a[z].P + b.Q)), [a.\mathbf{0}][\underline{i}](b[j].(a.P + b[w].Q))) = \\ & \text{fcc}(b[j].(a[z].P + b.Q), b[j].(a.P + b[w].Q)) = \text{fcc}(a[z].P + b.Q, a.P + b[w].Q) = \text{True} \end{aligned}$$

◇

Lemma 4.8. *Function fcc above is total for each Y and Z obtained via communication actions from a common X .*

Proof. We proceed by structural induction on X , with a case analysis on the rules used to derive the two transitions.

$X = \pi.P$: the only possibility here is that the prefix is executed, with two different keys, this case is covered by the first clause;

$X = [P](Q)$: here the only possibility is that the first component is executed, this case is covered by the first clause for timeout;

$X = X_1 + X_2$: this case is covered by the fourth clause;

$X = X_1 \parallel X_2$: this case is covered by the third clause;

$X = X_1 \setminus a$: this case is covered by the fifth clause;

$X = A$: constant A has a definition $A \stackrel{def}{=} P$, hence the proof for P applies. Note that in this case termination by structural induction is not granted, but termination is ensured anyway since recursion is guarded;

$X = \mathbf{0}$: since X cannot take any communication action, this case never applies;

$X = \rho[i].X_1$: this case is covered by the sixth clause;

$X = [X_1][\underline{j}](X_2)$: this case is covered by the one but last clause;

$X = [X_1][\underline{i}](X_2)$: this case is covered by the last clause. □

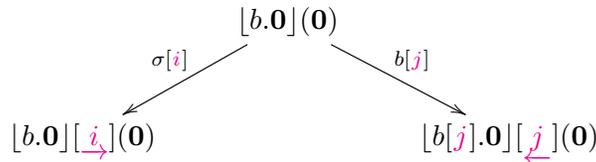
We now define a notion of conflict, and independence as its negation.

Definition 4.9 (Conflict and independence). Given a reachable revTPL configuration X , two cointitial transitions $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ are conflicting, if and only if one of the following conditions holds:

- (1) $t : X \xrightarrow{\sigma[i]} Y$ and $s : X \xrightarrow{\alpha[j]} Z$ or vice versa;
- (2) $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ are in forward communication conflict;
- (3) $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ with $j \leq_Y i$ or vice versa;
- (4) $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ with $i = j$.

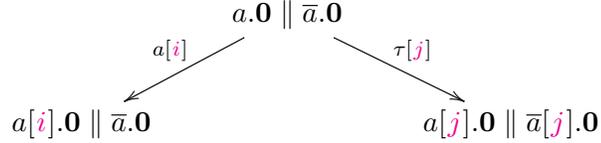
Transitions t and s are *independent*, written $t \mathcal{I} s$, if they are not conflicting.

Note that the conflict relation is reflexive and symmetric, hence independence is irreflexive and symmetric. The first clause of Definition 4.9 tells us that a delay cannot be swapped with a communication action. Consider configuration $[b.\mathbf{0}](\mathbf{0})$:



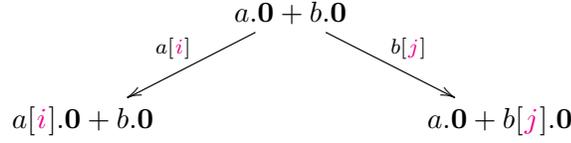
Transitions $\sigma[i]$ and $b[j]$ are in conflict: they cannot be swapped since action b is no longer possible after action σ , and vice versa.

The second case of Definition 4.9 forbids either a same prefix or prefixes in different branches of a same choice operator to be consumed by the two transitions. For example let us consider the configuration $a.\mathbf{0} \parallel \bar{a}.\mathbf{0}$. The left configuration could execute an action $a[i]$ while the entire configuration could synchronise by doing a $\tau[j]$, as depicted below:



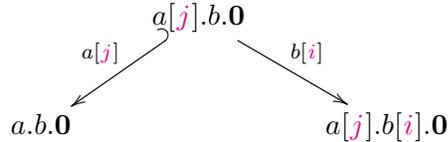
It is clear, from the example above, that the two actions cannot commute.

Another example of conflicting transitions captured by case 2 of Definition 4.9 is when transitions consume prefixes in different branches of a same choice operator. For example, let us consider the configuration $a.\mathbf{0} + b.\mathbf{0}$. The left branch can do an action $a[i]$ while the right one an action $b[j]$, as follows:



and again it is clear that these two transitions cannot commute.

The third clause of Definition 4.9 dictates that two transitions are in conflict when a reverse step eliminates some causes of a forward step. For example, the configuration $a[j].b.\mathbf{0}$ can do a forward step with label $b[i]$ going to $a[j].b[i].\mathbf{0}$ or a backward one with label $a[j]$, as follows:



We have that $\text{po}(a[j].b[i].\mathbf{0}) = \{j < i\}$, hence the side condition is satisfied. Undoing $a[j]$ disables the action on b .

The last case of Definition 4.9 forbids two transitions to pick up the same key.

Notably, backward transitions are never in conflict, yet it is never the case that a backward time action and a backward communication action are enabled together, as shown by the following proposition.

Proposition 4.10. *Let X be a reachable revTPL configuration. Then it is never the case that $X \xrightarrow{\sigma[i]} X'$ and $X \xrightarrow{\alpha[j]} X''$.*

Proof. The proof is by structural induction on X . If X is standard there is nothing to prove. If X is a prefix $\rho[k].Y$ and Y is standard then only communication actions are possible if ρ is a communication action, only time actions otherwise. If Y is not standard then the thesis follows by inductive hypothesis, since only rule [ACT] is applicable. In the case of timeout, the thesis follows by noticing that at most one rule is applicable. In particular, for all the

rules the thesis follows by inductive hypothesis but for rule [STOUT], for which it follows directly. The other cases follow by inductive hypothesis. \square

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

Property 4.11 (Square Property - SP). Given a reachable **revTPL** configuration X and two coinital transitions $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ with $t \mathcal{I} s$ there exist two cofinal transitions $t' : Y \xrightarrow{\pi_2[j]} W$ and $s' : Z \xrightarrow{\pi_1[i]} W$.

Proof. Deferred to Appendix C. \square

Since both CCSK and TPL are sub-calculi of **revTPL** as discussed in Section 3, then the notions of conflict and independence above induce analogous notions on CCSK and TPL. To the best of our knowledge, no such notion exists for TPL. Notions of conflict and independence (dubbed concurrency) for CCSK have been recently proposed in [Aub22], but they rely on extended labels while we define them on standard ones.

4.2. Causal consistency. We can now prove causal consistency, using the theory in [LPU20]. The theory in [LPU20] ensures that causal consistency follows from SP, already discussed, and two other properties: BTI (Backward Transitions are Independent) and WF (Well-Foundedness). BTI generalises the concept of backward determinism used for reversible sequential languages [YG07]. It specifies that two backward transitions from a same configuration are always independent.

Property 4.12 (Backward Transitions are Independent - BTI). Given a reachable **revTPL** configuration X , any two distinct coinital backward transitions $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ are independent.

BTI property trivially holds since (as mentioned above) by looking at the definition of conflicting and independent transitions (Definition 4.9) 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 configurations have a finite past.

Property 4.13 (Well-Foundedness - WF). Let X_0 be a reachable **revTPL** configuration. Then there is no infinite sequence such that $X_i \xrightarrow{\pi_i[j_i]} X_{i+1}$ for all $i = 0, 1, \dots$

Proof. WF follows since each backward transition removes a key. Given that the number $|\mathbf{keys}(X)|$ of keys in X is finite, only a finite amount of backward steps can be taken. \square

The Parabolic Lemma [DK04, Lemma 11], stated below, 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 4.14 (Parabolic Lemma (PL) [DK04, Lemma 11] property). An LTS satisfies the Parabolic Lemma iff for any path χ , there exist two forward-only paths ω, ω' such that $\chi \asymp \underline{\omega}\omega'$ and $|\omega| + |\omega'| \leq |\chi|$.

We can now prove our main results thanks to the proof schema of [LPU20].

Proposition 4.15 (cf. Proposition 3.4 [LPU20]). *Suppose BTI and SP hold, then PL holds.*

Proposition 4.16 (cf. Proposition 3.6 [LPU20]). *Suppose WF and PL hold, then CC holds.*

As a corollary of PL, reachable states are reachable via forward-only paths (cf. [LPU20]).

Corollary 4.17. *A configuration X is reachable iff there exists a process P and a forward-only path $P \rightarrow^* X$.*

Proof. From PL, by noticing that the backward path is empty since P cannot take backward actions. \square

The general theory above can help us in proving specific properties of revTPL, as we show below.

We have considered in this paper a global notion of time, as shown by the following theorem.

Theorem 4.18. *For each reachable revTPL configuration X , the restriction of \leq_X to keys attached to time actions is a total order.*

Proof. From Corollary 4.17 we have that there exist a process P and a forward-only path $P \rightarrow^* X$. Take two arbitrary keys i and j attached to time actions. Let i be the first one to occur in $P \rightarrow^* X$, and $X_1 \xrightarrow{\sigma[i]} X_2$ the transition introducing it (note that each step introduces a key). Since the path is forward, key j will be attached to some configuration which is standard in X_2 (or to a σ_\perp just before a standard configuration). We show by induction on the derivation of $X_1 \xrightarrow{\sigma[i]} X_2$ that this implies $i < j$. The thesis will follow. We have a case analysis on the last applied rule.

The cases of rules [PACT], [RACT] and [IDLE] follow from the definition of po on prefix. The cases of rules [ACT], [HIDE] and [CONST] follow by induction. The cases for timeout are similar, noticing that j could only be attached to the selected configuration. For parallel composition, only rule [SYNW] needs to be considered, and the thesis follows by inductive hypothesis. Similarly, for choice only rule [CHOW] needs to be considered, and the thesis follows by induction as well. \square

As shown above, time actions are never independent, and only communication actions can be. Also, since time actions do not commute with communication actions (cf. clause 1 in Definition 4.9) then each communication action is bound to be executed between two fixed time actions.

One may wonder whether the global notion of time described above is too strict. This is a very good question, and indeed we plan in future work to investigate different notions of causality for TPL, which will induce a different causal-consistent reversible extension.

We show here just that dropping the $\sigma_\perp[i]$ prefix, which ensures time actions are recorded also by untimed configurations, would not solve the issue. We have pursued this approach in [BLMY22], but it leads to violations of the Loop Lemma and the Parabolic Lemma, two main properties in the causal-consistency theory, as shown by the following example.

Example 4.19. Let us consider the configuration $X = \sigma.a.\mathbf{0} \parallel b.\sigma.\mathbf{0}$ and the following execution:

$$X \xrightarrow{\sigma[i]} \sigma[i].a.\mathbf{0} \parallel b.\sigma.\mathbf{0} \xrightarrow{b[j]} \sigma[i].a.\mathbf{0} \parallel b[j].\sigma.\mathbf{0} \xrightarrow{\sigma[k]} \sigma[i].a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} = Z$$

Now from Z we can undo the time actions $\sigma[i]$ and $\sigma[k]$ as follows:

$$Z \xrightarrow{\sigma[i]} \sigma.a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} = Z_1 \xrightarrow{\sigma[k]} \sigma.a.\mathbf{0} \parallel b[j].\sigma.\mathbf{0} = Z_2$$

Now let us focus on the last transition. According to the Loop Lemma (Lemma 4.1) we can reach Z_1 from Z_2 by doing a forward time action, that is $Z_2 \xrightarrow{\sigma[k]} Z_1$, but this is impossible as

$$\sigma.a.\mathbf{0} \parallel b[j].\sigma.\mathbf{0} \xrightarrow{\sigma[k]} \sigma[k].a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} \neq Z_1$$

Also, the Parabolic Lemma fails. Indeed if we consider the path which leads to Z_1 , according to the Parabolic Lemma, we can rewrite this path as a sequence of backward transitions followed by forward ones. If from Z_1 we undo all the actions and try to reach it by using just forward actions we fail since:

$$\begin{aligned} Z_1 = \sigma.a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} &\xrightarrow{\sigma[k]} \xrightarrow{b[j]} \sigma.a.\mathbf{0} \parallel b.\mathbf{0} \\ &\xrightarrow{b[j]} \sigma.a.\mathbf{0} \parallel b[j].\sigma.\mathbf{0} \xrightarrow{\sigma[k]} \sigma[k].a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} \neq Z_1 \end{aligned}$$

By using $\sigma_{\perp}[i]$ prefixes we impose a total order among time actions, as shown in Theorem 4.18, as follows:

$$\begin{aligned} X \xrightarrow{\sigma[i]} \sigma[i].a.\mathbf{0} \parallel \sigma_{\perp}[i].b.\sigma.\mathbf{0} &\xrightarrow{b[j]} \sigma[i].a.\mathbf{0} \parallel \sigma_{\perp}[i].b[j].\sigma.\mathbf{0} \xrightarrow{\sigma[k]} \\ &\sigma[i].\sigma_{\perp}[k].a.\mathbf{0} \parallel b[j].\sigma[k].\mathbf{0} = Y \end{aligned}$$

Now from Y we cannot undo the time action $\sigma[i]$, since now we need to undo action $\sigma[k]$ first. With this machinery in place, we enforce a strict notion of causality in TPL, but we have been able to successfully build a causal-consistent reversible extension. \diamond

5. CONCLUSION, RELATED AND FUTURE WORK

The main contribution of this paper is the study of the interplay between causal-consistent reversibility and time. A reversible semantics for TPL cannot be automatically derived using well-established frameworks [PU07, LM20], 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 [PU07] not applicable. To faithfully capture patient actions in a reversible semantics we introduced σ_{\perp} prefixes. Another peculiarity of TPL is the timeout operator $[P](Q)$, which can be seen as a choice operator whose left branch has priority over the right one. Indeed, if P can do a τ action then Q can not execute and it is discarded. Although we have been able to use the static approach to reversibility [PU07], adapting it to our setting has been challenging for the aforementioned reasons. Notably, our results have a double interpretation: as an extension of CCSK [PU07] with time, and as a reversible extension of TPL [HR95]. As a side result, by focusing on the two fragments, we derive notions of independence and conflict for CCSK and TPL.

Other process algebras. As said by Baeten and Bergstra “*Adding real time features can be done in many ways and it is impossible to explore all options in a single paper*” [BB91]. The literature of timed process calculi is indeed rich. Thus, we only give an outline of the main approaches with the purpose of reflecting on the applicability of our results to different time approaches. Besides TPL [HR95], considered in this paper, a non-exhaustive list of alternative formalisms includes timed CSP [RR86], temporal CCS [MT90], timed CCS [Yi91], real-time ACP [BB91], urgent LOTOS [BL92], CIPA [AM96], ATP [NS94], TIC [QdFA93], PAFAS [CVJ02], and mCRL2 [GM14].

To simplify the discussion, we build on the categorisation in [BM23] and focus our comparison on the following time-related design choices:

Separated vs integrated semantics: In the first case, actions are instantaneous and time only passes in between actions; hence, functional behaviour and time are orthogonal. In the second case, every action takes a certain amount of time to be performed and time passes only due to action execution; hence, functional behaviour and time are integrated.

Relative time vs absolute delays: In the first case, each delay refers to the time instant of the previous observation. In the second case, all delays refer to the starting time of the system’s execution.

Global clock vs local clocks: In the first case, a single clock governs the pace of time passing in the system. In the second case, several clocks associated with the various system parts may have different views of the pace of time. If a model allows processes to have local clocks but time flows at the same pace for all of them (even if they hold different values due to resets, as in the case e.g. of Timed Automata [AD94]) we still classify the model as a global clock model.

Eager vs lazy vs maximal progress: There are several interpretations of when a communication action can be executed or delayed. Eager semantics enforce actions to be performed as soon as they become enabled, i.e., without any delay, thereby implying that their execution is urgent. On the other hand, laziness allows the execution of an action to be delayed even if the action is enabled. Maximal progress is eager for internal actions and lazy otherwise: actions can be delayed only if they are waiting to synchronise with some external partner which is not yet available. Some calculi have primitives for both eager and lazy actions, so each action can be either lazy or eager.

Table 1 illustrates how the aforementioned timed calculi position with respect to the four criteria above. Most of the formalisms we have reviewed combine **separated semantics**, **relative time**, and **global clock**. The main difference is the urgency (or lack thereof) of communication actions with respect to time actions. ATP [NS94], temporal CCS [MT90], and PAFAS [CVJ02] allow actions to happen at any time within the prescribed intervals (e.g., later than when they become ready to execute). Instead, timed CSP [RR86] and timed CCS [Yi91] share the same approach we adopted in this paper, inherited from TPL [HR95]: actions are normally lazy, unless they are silent in which case they are eager (maximal progress). A more general approach is the one of urgent LOTOS [BL92], which provides primitives for urgent actions and primitives for non-urgent actions, hence enabling one to decide the semantics of each specific action. The remaining formalisms have **integrated semantics** combined with **absolute time**. In mCRL [GM14], CIPA [QdFA93], and TIC [QdFA93], the transition relation models both execution of actions and time elapsing

	semantics	time	clocks	actions
ATP [NS94]	separated	relative	global	lazy
temporal CCS [MT90]	separated	relative	global	lazy
PAFAS [CVJ02]	separated	relative	global	lazy
TPL [HR95]	separated	relative	global	maximal progress
timed CSP [RR86]	separated	relative	global	maximal progress
timed CCS [Yi91]	separated	relative	global	maximal progress
urgent LOTOS [BL92]	separated	relative	global	either
mCRL2 [GM14]	integrated	absolute	global	lazy
CIPA [AM96]	integrated	absolute	local	eager
TIC [QdFA93]	integrated	absolute	local	either

Table 1: Semantics can be separated or integrated; time can be relative or absolute; clocks can be global or local; actions can be eager, lazy, either of them, or maximal progress.

(integrated semantics).² In all the three calculi, time is specified from the beginning of the computation (absolute time). While mCRL relies on a global clock, CIPA and TIC allow parallel processes to go ‘out of sync’ (local clocks). In CIPA, global time synchronisation is only required for causally dependent actions (it has to be re-established before two processes can communicate with each other). TIC uses an ‘age’ function to record discrepancies between the time of parallel processes. mCRL has no silent actions, and time idling and communication actions can happen at any time, after they become ready (lazy). In CIPA, the timing of an action needs to exactly match its prescription so the action happens as soon as it is ready (eager). TIC allows delays of exact amounts of time (urgent/eager) as well as delays of times within an interval (lazy).

The application of our approach using integrated semantics and/or absolute time should not present any particular challenge. In fact, separated and integrated semantics have been shown to be equivalent [BCT16] (i.e., they can be encoded into each other preserving weak barbed bisimilarity). Similarly for absolute instead of relative time thanks to the equivalence given in [Cor00].

Extending our framework to local clocks (e.g., as in CIPA) would be interesting but non-trivial in our integrated semantics. It may require us to record some live information on the different time perspective of parallel processes to rule out unwanted interleavings. An alternative could be to exploit the encoding of [BCT16] from TCCS to CIPA, and see whether the semantics is still preserved while considering reversible behaviours.

Building from the conference version of this article [BLMY22], the work in [BM23] has shown that our approach would apply also to a semantics with only eager actions and to a semantics with only lazy actions. However, the applicability of our approach to a scenario where each action can be statically set to be either lazy or eager (the ‘either’ option in the ‘action’ column of Table 1) needs to be further investigated.

Alternative timed formalisms. Timed Petri nets are a relevant tool for analysing real-time systems. A step towards the analysis of real-time systems would be to encode revTPL

²The transition relation of mCRL does also feature an idling relation, but this does not lead to any follow-up state and is just for final states.

into timed Petri nets [ZFH01] extended with reversibility, by building on the encoding of reversible CCS into reversible Petri nets [MMP21b]. Also, we could think of encoding revTPL in timed automata [ACD93] extended with reversibility. Another possibility would be to study the extension of a monitored timed semantics for multiparty session types, as the one of [NBY17], with reversibility [MP21].

Maximal progress of TPL (as well as revTPL) has connections with Markov chains [BH00]. For instance, in a stochastic process algebra, the process

$$\tau.P + (\lambda).Q$$

(where λ is a rate) will not be delayed since τ is instantaneously enabled. This is similar to 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 [BCT16]. Further investigation on the relation between our work and [BM20], studying reversibility in Markov chains, is left for future work. The treatment of passage of time shares some similarities with broadcast [Mez18] as well: time actions affect parallel components in the same way.

Future directions. We have just started our research quest towards a reversible timed semantics. Beyond considering local notions of time, as discussed after Theorem 4.18, a first improvement would be to add an explicit rollback operator, as in [LMSS11], 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 [HR95], including an axiomatization. A further step could be to adapt such behavioural theory to account for reversibility, by combining it with the one for CCSK developed in [LP21]. However, the fact that reversibility breaks Milner’s expansion law may limit the power of the axiomatisation. Also, we could consider studying more complex temporal operators [NS91]. In TPL time is discrete, and the language abstracts away from how time is represented. Indeed, the idling prefix σ is meant to await one cycle of clock. A more fine-grained treatment of time in CCS was proposed in Timed CCS (TCCS) [Yi90, Yi91]. In TCCS it is possible to express a process, say P , which awaits 3 time units directly by:

$$\epsilon(3).P$$

Now the process above, in principle, can be rendered in TPL as the process $\sigma.\sigma.\sigma.P$ by assuming that a cycle of clock lasts one time unit. But this is only possible if we consider TCCS with discrete time. Even if we restrict ourselves to discrete time, encoding the $\epsilon(\cdot)$ operator in TPL would be troublesome to treat (from a reversible point of view) as a single step has to be matched by several ones. Also, TCCS obeys to *time additivity* (two actions taking times t_1 and t_2 can be turned into a single action taking time $t_1 + t_2$), while TPL does not. As shown in [BM23], time additivity poses a problem with our approach: in presence of time additivity, the proof schema proposed in [LPU20] does not hold anymore. In particular, because of time additivity BTI does not hold anymore and Loop Lemma has to be formalised in a weaker form. Hence, one has to redo all the proofs. For all these reasons, it will not be straightforward to adapt the approach in this paper to deal with TCCS.

Prospective applications. As discussed above, this work is a first step towards an analysis of reversible real-time systems and it has the purpose of clarifying the relationship between reversibility and time. Although the contribution of this work is theoretical, we envisage a potential application to debugging of real-time Erlang code. More concretely, we would like to extend CauDER [LNPV18, GV21, Cau22], the only causal-consistent reversible debugger for a (fragment of a) real programming language we are aware of. The purpose of the extension would be to support *timed* Erlang programs. To this end we would first need to extend the reversible semantics of Erlang in [LLS⁺22, FLS21] with a notion of time, imported from the present work, so to support constructs such as ‘after’ and ‘sleep’, as used, e.g., in our Example 0.1. The ‘after’ (i.e., timeout) construct, in particular, is very common in the Erlang programming practice. Even if Erlang timeouts are close to TPL ones, there are a number of challenges to be faced. First, Erlang communication is asynchronous, unlike revTPL. Second, and more importantly, Erlang delays can be explicit in the code, as in our Example 0.1, but they can also be generated by network delays or long computations. Therefore, in order to enable reversible debugging of timed programs, one needs to pair the code with a model, possibly computed in an automated way, that describes the delays that are likely to occur in the system of interest. The development of this prospective application goes beyond the scope of the formal setting given in the current work.

Another possible application is to bring our theory to timed Rebecca [KSS⁺15] which is a timed actor based language with model checking support. This would enable us to exploit model checking for reversible behaviours.

REFERENCES

- [ACD93] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104(1):2–34, 1993. doi:10.1006/inco.1993.1024.
- [AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. doi:10.1016/0304-3975(94)90010-8.
- [AM96] Luca Aceto and David Murphy. Timing and causality in process algebra. *Acta Informatica*, 33(4):317–350, 1996. doi:10.1007/s002360050047.
- [Aub22] Clément Aubert. Concurrency in reversible concurrent calculi. In Claudio Antares Mezzina and Krzysztof Podlaski, editors, *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, volume 13354 of *Lecture Notes in Computer Science*, pages 146–163. Springer, 2022. doi:10.1007/978-3-031-09005-9_10.
- [BB91] Jos C. M. Baeten and J. A. Bergstra. Real time process algebra. *Form. Asp. Comput.*, 3(2):142–188, jun 1991. doi:10.1007/BF01898401.
- [BCT16] Marco Bernardo, Flavio Corradini, and Luca Tesei. Timed process calculi with deterministic or stochastic delays: Commuting between durational and durationless actions. *Theor. Comput. Sci.*, 629:2–39, 2016. doi:10.1016/j.tcs.2016.02.022.
- [Ber05] Ivan Cibrario Bertolotti. Real-time embedded operating systems. In Richard Zurawski, editor, *Embedded Systems Handbook*. CRC Press, 2005. doi:10.1201/9781420038163.ch11.
- [BH00] Ed Brinksma and Holger Hermanns. Process algebra and Markov chains. In Ed Brinksma, Holger Hermanns, and Joost-Pieter 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. doi:10.1007/3-540-44667-2_5.
- [BJCC13] Tom Britton, Lisa Jeng, Graham Carver, and Paul Cheak. Reversible debugging software “quantify the time and cost saved using reversible debuggers”, 2013. URL: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf>.
- [BL92] Tommaso Bolognesi and Ferdinando Lucidi. Lotos-like process algebras with urgent or timed interactions. In K.R. Parker and G.A. Rose, editors, *Formal Description Techniques, IV*, IFIP

- Transactions C: Communication Systems, pages 249–264. Elsevier, Amsterdam, 1992. doi: 10.1016/B978-0-444-89402-1.50027-8.
- [BLMY22] Laura Bocchi, Ivan Lanese, Claudio Antares Mezzina, and Shoji Yuen. The reversible temporal process language. In Mohammad Reza Mousavi and Anna Philippou, editors, *FORTE 2022*, volume 13273 of *Lecture Notes in Computer Science*, pages 31–49. Springer, 2022. doi:10.1007/978-3-031-08679-3_3.
- [BM20] Marco Bernardo and Claudio Antares Mezzina. Towards bridging time and causal reversibility. In Alexey Gotsman and Ana 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. doi:10.1007/978-3-030-50086-3_2.
- [BM23] Marco Bernardo and Claudio Antares Mezzina. Causal reversibility for timed process calculi with durationless lazy/eager actions and time additivity. In Laure Petrucci and Jeremy Sproston, editors, *Formal Modeling and Analysis of Timed Systems. FORMATS 2023*, volume 14138 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2023. doi:10.1007/978-3-031-42626-1_2.
- [Cau22] CauDer repository. Available at <https://github.com/mistupv/cauder>, 2022.
- [CKV13] Ioana Cristescu, Jean Krivine, and Daniele 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. doi:10.1109/LICS.2013.45.
- [Cor00] Flavio Corradini. Absolute versus relative time in process algebras. *Inf. Comput.*, 156(1-2):122–172, 2000. doi:10.1006/inco.1999.2821.
- [CVJ02] Flavio Corradini, Walter Vogler, and Lars Jenner. Comparing the worst-case efficiency of asynchronous systems with PAFAS. *Acta Informatica*, 38(11/12):735–792, 2002. doi:10.1007/s00236-002-0094-3.
- [DK04] Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004. doi:10.1007/978-3-540-28644-8_19.
- [FGP12] Julie Street Fant, Hassan Gomaa, and Robert G. Pettit IV. A comparison of executable model based approaches for embedded systems. In He Zhang, Liming Zhu, and Ihor Kuz, editors, *Second International Workshop on Software Engineering for Embedded Systems, SEES 2012*, pages 16–22. IEEE, 2012. doi:10.1109/SEES.2012.6225484.
- [FLS21] Giovanni Fabbretti, Ivan Lanese, and Jean-Bernard Stefani. Causal-consistent debugging of distributed Erlang programs. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2021. doi:10.1007/978-3-030-79837-6_5.
- [GLM14] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *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. Algebraic Methods Program.*, 88:99–120, 2017. doi:10.1016/j.jlamp.2016.09.003.
- [GM14] Jan Friso Groote and Mohammad Reza Mousavi. *Modeling and Analysis of Communicating Systems*. The MIT Press, 2014.
- [Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Fifth Symposium on Reliability in Distributed Software and Database Systems, SRDS 1986, Los Angeles, California, USA, January 13-15, 1986, Proceedings*, pages 3–12. IEEE Computer Society, 1986.
- [GV21] Juan José González-Abril and Germán Vidal. Causal-consistent reversible debugging: Improving CauDer. In José F. Morales and Dominic A. Orchard, editors, *Practical Aspects of Declarative Languages - 23rd International Symposium, PADL*, volume 12548 of *Lecture Notes in Computer Science*, pages 145–160. Springer, 2021. doi:10.1007/978-3-030-67438-0_9.
- [GZT20] Mageed Ghaleb, Hossein Zolfagharinia, and Sharareh 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. doi:10.1016/j.cor.2020.105031.
- [HR95] Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995. doi:10.1006/inco.1995.1041.

- [Koo10] Philip Koopman. *Better Embedded System Software*. Drumnadrochit Press, 2010. URL: <http://koopman.us/book.html>.
- [KSS⁺15] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi, and Mohammad-Javad Izadi. Timed rebecca schedulability and deadlock freedom analysis using bounded floating time transition system. *Sci. Comput. Program.*, 98:184–204, 2015. doi:10.1016/j.scico.2014.07.005.
- [LLS⁺22] Pietro Lami, Ivan Lanese, Jean-Bernard Stefani, Claudio Sacerdoti Coen, and Giovanni Fabbretti. Reversibility in Erlang: Imperative constructs. In Claudio Antares Mezzina and Krzysztof Podlaski, editors, *Reversible Computation - 14th International Conference, RC 2022, Urbino, Italy, July 5-6, 2022, Proceedings*, volume 13354 of *Lecture Notes in Computer Science*, pages 187–203. Springer, 2022. doi:10.1007/978-3-031-09005-9_13.
- [LM20] Ivan Lanese and Doriana Medic. A general approach to derive uncontrolled reversible semantics. In Igor Konnov and Laura 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. doi:10.4230/LIPICs.CONCUR.2020.33.
- [LMM21] Ivan Lanese, Doriana Medic, and Claudio Antares Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, 58(1-2):1–34, 2021. doi:10.1007/s00236-019-00346-6.
- [LMS16] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π -calculus. *Theor. Comput. Sci.*, 625:25–84, 2016. doi:10.1016/j.tcs.2016.02.019.
- [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 2011 - Concurrency Theory - 22nd International Conference*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011. doi:10.1007/978-3-642-23217-6_20.
- [LMT14] Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent reversibility. *Bull. EATCS*, 114, 2014. URL: <http://bulletin.eatcs.org/index.php/beatcs/article/view/305>.
- [LNPV18] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. CauDER: A causal-consistent reversible debugger for Erlang. In John P. Gallagher and Martin Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS 2018, Nagoya, Japan, May 9-11, 2018, Proceedings*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer, 2018. doi:10.1007/978-3-319-90686-7_16.
- [LP21] Ivan Lanese and Iain C. C. Phillips. Forward-reverse observational equivalences in CCSK. In Shigeru Yamashita and Tetsuo 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. doi:10.1007/978-3-030-79837-6_8.
- [LPU20] Ivan Lanese, Iain C. C. Phillips, and Irek Ulidowski. An axiomatic approach to reversible computation. In Jean Goubault-Larrecq and Barbara König, editors, *FOSSACS 2020*, volume 12077 of *Lecture Notes in Computer Science*, pages 442–461. Springer, 2020. doi:10.1007/978-3-030-45231-5_23.
- [LY97] Kim Guldstrand Larsen and Wang Yi. Time-abstracted bisimulation: Implicit specifications and decidability. *Inf. Comput.*, 134(2):75–101, 1997. doi:10.1006/inco.1997.2623.
- [Mez18] Claudio Antares Mezzina. On reversibility and broadcast. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018*, volume 11106 of *Lecture Notes in Computer Science*, pages 67–83. Springer, 2018. doi:10.1007/978-3-319-99498-7_5.
- [Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. doi:10.1007/3-540-10235-3.
- [MMP21a] Hernán C. Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. A distributed operational view of reversible prime event structures. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2021*, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470623.
- [MMP21b] Hernán C. Melgratti, Claudio Antares Mezzina, and G. Michele Pinna. Towards a truly concurrent semantics for reversible CCS. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021*, volume 12805 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2021. doi:10.1007/978-3-030-79837-6_7.
- [MMPY20] Doriana Medic, Claudio Antares Mezzina, Iain C. C. Phillips, and Nobuko Yoshida. A parametric framework for reversible π -calculi. *Inf. Comput.*, 275:104644, 2020. doi:10.1016/j.ic.2020.104644.

- [MMU19] Hernán C. Melgratti, Claudio Antares Mezzina, and Irek Ulidowski. Reversing P/T nets. In Hanne Riis Nielson and Emilio Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2019. doi:10.1007/978-3-030-22397-7_2.
- [MP20] Raffaele Mauro and Andrea 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. doi:10.1016/j.jtte.2019.09.007.
- [MP21] Claudio Antares Mezzina and Jorge A. Pérez. Causal consistency for reversible multiparty protocols. *Logical Methods in Computer Science*, Volume 17, Issue 4, October 2021. doi:10.46298/lmcs-17(4:1)2021.
- [MT90] Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, pages 401–415, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg. doi:10.1007/BFb0039073.
- [NBY17] Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Aspects Comput.*, 29(5):877–910, 2017. doi:10.1007/s00165-017-0420-8.
- [NMV90] Rocco De Nicola, Ugo Montanari, and Frits W. Vaandrager. Back and forth bisimulations. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990. doi:10.1007/BFb0039058.
- [NS91] Xavier Nicollin and Joseph Sifakis. An overview and synthesis on timed process algebras. In Kim Guldstrand Larsen and Arne Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV '91*, volume 575 of *Lecture Notes in Computer Science*, pages 376–398. Springer, 1991. doi:10.1007/3-540-55179-4_36.
- [NS94] Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: Theory and application. *Information and Computation*, 114(1):131–178, 1994. doi:10.1006/inco.1994.1083.
- [PP18] Anna Philippou and Kyriaki Psara. Reversible computation in Petri nets. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, volume 11106 of *Lecture Notes in Computer Science*, pages 84–101. Springer, 2018. doi:10.1007/978-3-319-99498-7_6.
- [PU07] Iain C. C. Phillips and Irek Ulidowski. Reversing algebraic process calculi. *J. Log. Algebraic Methods Program.*, 73(1-2):70–96, 2007. doi:10.1016/j.jlap.2006.11.002.
- [PUY12] Iain C. C. Phillips, Irek Ulidowski, and Shoji Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In Robert Glück and Tetsuo Yokoyama, editors, *Reversible Computation, 4th International Workshop, RC 2012*, volume 7581 of *Lecture Notes in Computer Science*, pages 218–232. Springer, 2012. doi:10.1007/978-3-642-36315-3_18.
- [QdFA93] Juan Quemada, David de Frutos, and Arturo Azcorra. Tic: A timed calculus. *Formal Aspects of Computing*, 5(3):224–252, 1993. doi:10.1007/BF01211556.
- [RR86] George M. Reed and Andrew W. Roscoe. A timed model for communicating sequential processes. In Laurent Kott, editor, *Automata, Languages and Programming*, pages 314–323, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
- [Viz20] Mike Vizard. Report: Debugging efforts cost companies \$61b annually, 2020. URL: <https://devops.com/report-debugging-efforts-cost-companies-61b-annually/>.
- [YG07] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In G. Ramalingam and Eelco Visser, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 144–153. ACM, 2007. doi:10.1145/1244381.1244404.
- [Yi90] Wang Yi. Real-time behaviour of asynchronous agents. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR '90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 27-30, 1990, Proceedings*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer, 1990. doi:10.1007/BFb0039080.
- [Yi91] Wang Yi. CCS + time = an interleaving model for real time systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *Automata, Languages and*

Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1991. doi:10.1007/3-540-54233-7_136.

- [ZFH01] Armin Zimmermann, Jörn Freiheit, and Günter 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. doi:10.1109/IPDPS.2001.925065.

APPENDIX A. BACKGROUND: CCS, CCSK AND TPL

In this section we present the full syntax and semantics of CCS, TPL and CCSK, taken respectively from [Mil80], [HR95] and [PU07].

A.1. CCS: Calculus of Communicating Systems. The Calculus of Communicating Systems is a process calculus introduced by Milner [Mil80]. We let \mathcal{A} be the set of action names a , $\bar{\mathcal{A}}$ the set of action conames \bar{a} . We use α to range over a , \bar{a} and internal actions τ . We assume $\bar{\bar{a}} = a$. We let $\mathcal{A}^\tau = \mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau\}$. The syntax of CCS is reported in Figure 5. A process can be an action *prefix* $\alpha.P$, that can perform an action α and continue as P , a non-deterministic *choice* $P + Q$ among two processes, a *parallel* composition $P \parallel Q$ of two processes, the *restriction* $P \setminus a$, which acts as the process P , but actions on a are forbidden, the constant *identifier* A and the *inactive* process $\mathbf{0}$. The semantics of CCS is given by the labelled transition system (LTS) $(\mathcal{P}, \mathcal{A}^\tau, \rightarrow_\tau)$, where \mathcal{P} is the set of all CCS processes, \mathcal{A}^τ is the set of labels and \rightarrow_τ is the least transition relation induced by the rules in Figure 6.

A.2. CCSK: CCS with communication keys. CCS with communication keys (CCSK) is a reversible extension of CCS obtained by applying to CCS the approach in [PU07]. The key idea of this approach is to make all the dynamic operators (such as prefix and non-deterministic choice) static and to decorate prefixes with freshly created identifiers, dubbed communication keys, when they are executed. The syntax of CCSK is reported in Figure 7, where the addition with respect to the syntax of CCS (Figure 5) is enclosed in grey boxes. With respect to CCS, in CCSK a prefix α can be decorated with a communication key i , to indicate the fact that the prefix has been executed. As one can see, CCSK (reversible) configurations are built on top of CCS processes.

The semantics of CCSK is given by two LTSs, the *forward* one $(\mathcal{X}^k, \mathcal{A}^\tau \times \mathcal{K}, \rightarrow_k)$ and the *backward* one $(\mathcal{X}^k, \mathcal{A}^\tau \times \mathcal{K}, \hookrightarrow_k)$, where \mathcal{X}^k is the set of CCSK configurations, \mathcal{K} is the set of keys and \rightarrow_k and \hookrightarrow_k are the least transition relations induced by the rules in Figure 8 and Figure 9, respectively. Since CCSK is reversible, for each forward rule there exists a corresponding backward one. The two LTSs use two functions, $\text{std}(X)$ and $\text{keys}(X)$. Intuitively, function $\text{std}(X)$ states that a configuration X has no decorated prefixes (i.e., it does not contain any history information hence it is a CCS process), while function $\text{keys}(X)$ returns the set of keys of a given configuration.

(Processes) $P = \alpha.P \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A \mid \mathbf{0}$

Figure 5: Syntax of CCS

$$\begin{array}{c}
(\text{ACT}) \alpha.P \xrightarrow{\alpha}_t P \qquad (\text{SUM}_1) \frac{P \xrightarrow{\alpha}_t P'}{P + Q \xrightarrow{\alpha}_t P' + Q} \qquad (\text{SUM}_2) \frac{Q \xrightarrow{\alpha}_t Q'}{P + Q \xrightarrow{\alpha}_t P + Q'} \\
(\text{COM}_1) \frac{P \xrightarrow{\alpha}_t P'}{P \parallel Q \xrightarrow{\alpha}_t P' \parallel Q} \qquad (\text{COM}_2) \frac{Q \xrightarrow{\alpha}_t Q'}{P \parallel Q \xrightarrow{\alpha}_t P \parallel Q'} \qquad (\text{COM}_3) \frac{P \xrightarrow{\alpha}_t P' \quad Q \xrightarrow{\bar{\alpha}}_t Q'}{P \parallel Q \xrightarrow{\tau}_t P' \parallel Q'} \\
(\text{RES}) \frac{P \xrightarrow{\alpha}_t P' \quad \alpha \notin \{a, \bar{a}\}}{P \setminus a \xrightarrow{\tau}_t P' \setminus a} \qquad (\text{REC}) \frac{A \stackrel{\text{def}}{=} P \quad P \xrightarrow{\alpha}_t P'}{A \xrightarrow{\alpha}_t P'}
\end{array}$$

Figure 6: Semantics of CCS

(Processes) $P = \alpha.P \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A \mid \mathbf{0}$

(Configurations) $X = \alpha[i].X \mid X + Y \mid X \parallel Y \mid X \setminus a \mid P$

Figure 7: Syntax of CCSK

A.3. TPL: Timed Process Language. TPL [HR95] is an extension of CCS with time. Its syntax is reported in Figure 10, where the novelties w.r.t. CCS are enclosed in grey boxes. TPL adds to CCS two constructs to deal with time: the *idling* prefix $\sigma.P$ and the *timeout* operator $[P](Q)$. The process $\sigma.P$ acts as P after having waited one time unit, while the timeout $[P](Q)$ executes P (if possible) or Q (in case of a timeout). We indicate with \mathcal{A}^t the set $\mathcal{A} \cup \bar{\mathcal{A}} \cup \{\tau, \sigma\}$. The semantics of TPL is given via an LTS and two set of rules: rules for actions (Figure 11, similar to the CCS rules with the addition of rule THEN), and rules for time (Figure 12), which regulate the behaviour of the temporal operators $\sigma.P$ and $[P](Q)$ and the passage of time in the system (e.g., in a parallel composition). Hence, the semantics of TPL is given by the LTS $(\mathcal{P}^t, \mathcal{A}^t, \rightarrow_t)$, where \mathcal{P}^t is the set of TPL processes and \rightarrow_t is the least relation induced by the rules in Figures 11 and 12.

APPENDIX B. ENCODING OF NEGATIVE PREMISES

In this section we show that there exists an encoding of the negative premises in the rules of Figure 2 and Figure 3 into decidable positive premises. To do so we compute all the enabled forward prefixes (i.e., barbs) of a configuration and form all the possible pairs of prefixes on the two sides of a parallel operator. We then check whether there exists a pair containing both an action and the corresponding co-action. The operator γ is inductively defined as follows:

$$\begin{array}{c}
\text{(ACT1)} \frac{\text{std}(X)}{\alpha.X \xrightarrow{\alpha[i]}_k \alpha[i].X} \qquad \text{(ACT2)} \frac{X \xrightarrow{\beta[j]}_k X' \quad i \neq j}{\alpha[i].X \xrightarrow{\beta[j]}_k \alpha[i].X'} \\
\text{(SUM1)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad \text{std}(Y)}{X + Y \xrightarrow{\alpha[i]}_k X' + Y} \qquad \text{(SUM2)} \frac{Y \xrightarrow{\alpha[i]}_k Y' \quad \text{std}(Y)}{X + Y \xrightarrow{\alpha[i]}_k X + Y'} \\
\text{(PAR1)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad i \notin \text{keys}(Y)}{X \parallel Y \xrightarrow{[i]}_k X' \parallel Y} \qquad \text{(PAR2)} \frac{Y \xrightarrow{\alpha[i]}_k Y' \quad i \notin \text{keys}(X)}{X \parallel Y \xrightarrow{\alpha[i]}_k X \parallel Y'} \\
\text{(PAR3)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad Y \xrightarrow{\bar{\alpha}[i]}_k Y'}{X \parallel Y \xrightarrow{\tau[i]}_k X \parallel Y'} \\
\text{(RES)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad \alpha \notin \{a, \bar{a}\}}{X \setminus a \xrightarrow{\alpha[i]}_k X' \setminus a} \qquad \text{(REC)} \frac{A \stackrel{def}{=} P \quad P \xrightarrow{\alpha[i]}_k X}{A \xrightarrow{\alpha[i]}_k X}
\end{array}$$

Figure 8: Forward LTS of CCSK

Definition B.1 (Synchronisation Operator).

$$\begin{array}{ll}
\gamma(\alpha.P) = \{\{\alpha\}\} & \gamma(\rho[i].X) = \gamma(X) \\
\gamma(X \parallel Y) = \gamma(X) \oplus \gamma(Y) & \gamma(X \setminus a) = \gamma(X) \setminus \{a\} \\
\gamma(X + Y) = \gamma(X) \cup \gamma(Y) \text{ if } \mathbf{nact}(X + Y) & \\
\gamma(X + Y) = \gamma(X) \text{ if } \neg \mathbf{nact}(X) & \gamma(X + Y) = \gamma(Y) \text{ if } \neg \mathbf{nact}(Y) \\
\gamma(\lfloor X \rfloor(Y)) = \gamma(X) & \gamma(\lfloor X \rfloor[\underline{i}](Y)) = \gamma(X) \\
\gamma(\lfloor X \rfloor[\underline{i}](Y)) = \gamma(Y) & \gamma(A) = \gamma(P) \text{ if } A \stackrel{def}{=} P \\
\gamma(\mathbf{0}) = \emptyset & \gamma(\sigma.P) = \emptyset
\end{array}$$

where $A \oplus B$ is defined as follows:

$$\{A_i \mid i \in I\} \oplus \{B_j \mid j \in J\} = \bigcup_{i \in I, j \in J} \{A_i \cup B_j\}$$

We can then define $X \xrightarrow{\tau}$ as:

$$X \xrightarrow{\tau} = \forall C \in \gamma(X). \forall c_i, c_j \in C. \bar{c}_i \neq c_j$$

The intuition behind the $\gamma(\cdot)$ operator is that for sequential processes (i.e., processes which have no top level \parallel) it computes a set of singletons of prefixes. Such a set represents the list of all the enabled forward prefixes, which could synchronise in a parallel composition. This is rendered by the rule $\gamma(X \parallel Y) = \gamma(X) \oplus \gamma(Y)$. In this case, via the operator \oplus , we compute all the possible pairs of such singletons. Let us note that if we have more than one

$$\begin{array}{c}
\text{(ACT1)} \frac{\text{std}(X)}{\alpha[i].X \xrightarrow{\alpha[i]}_k a.X} \qquad \text{(ACT2)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad i \neq j}{\alpha[i].X \xrightarrow{\alpha[i]}_k a.X} \\
\text{(SUM1)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad \text{std}(Y)}{X + Y \xrightarrow{\alpha[i]}_k X' + Y} \qquad \text{(SUM2)} \frac{Y \xrightarrow{\alpha[i]}_k Y' \quad \text{std}(X)}{X + Y \xrightarrow{\alpha[i]}_k X + Y'} \\
\text{(PAR1)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad i \notin \text{keys}(Y)}{X \parallel Y \xrightarrow{\alpha[i]}_k X' \parallel Y} \qquad \text{(PAR2)} \frac{Y \xrightarrow{\alpha[i]}_k Y' \quad i \notin \text{keys}(X)}{X \parallel Y \xrightarrow{\alpha[i]}_k X \parallel Y'} \\
\text{(PAR3)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad Y \xrightarrow{\bar{\alpha}[i]}_k Y'}{X \parallel Y \xrightarrow{\tau[i]}_k X \parallel Y'} \\
\text{(RES)} \frac{X \xrightarrow{\alpha[i]}_k X' \quad \alpha \notin \{a, \bar{a}\}}{X \setminus a \xrightarrow{\alpha[i]}_k X' \setminus a} \qquad \text{(REC)} \frac{A \stackrel{\text{def}}{=} P \quad X \xrightarrow{\alpha[i]}_k P}{P \xrightarrow{\alpha[i]}_k A}
\end{array}$$

Figure 9: Backward LTS of CCSK

$$\text{(Processes)} \quad P = \pi.P \mid [P](Q) \mid P + Q \mid P \parallel Q \mid P \setminus a \mid A \mid \mathbf{0} \quad (\pi = \alpha \mid \sigma)$$

Figure 10: Syntax of TPL

top level \parallel , say n , we will have a set of $(n + 1)$ -uples. Also, since we are using set-based operators, repetitions of prefixes will be dropped.

The γ operator just collects all the available prefixes, discarding the ones already executed, and the discarded branches. For example, let us consider the process $P = a + \bar{a}$ we have that $\gamma(P) = \{\{a\}, \{\bar{a}\}\}$. Synchronisation is induced by the parallel operator \parallel , hence when the γ operator meets a parallel composition, it computes all the possible pairs via the \oplus operator. We then have a possible synchronisation if there is a pair of the form $\{\alpha, \bar{\alpha}\}$. For example, if we take the process P above and we put it in parallel with a , the synchronisation operator will compute the following pairs

$$\gamma((a + \bar{a}) \parallel a) = \gamma(a + \bar{a}) \oplus \gamma(a) = \{\{a\}, \{\bar{a}\}\} \oplus \{\{a\}\} = \{\{a, \bar{a}\}, \{a, a\}\}$$

We can see that there exists one synchronisation pair. Furthermore, if we consider the process $(b + \bar{a}) \parallel a \parallel \bar{b}$ we have:

$$\begin{aligned}
\gamma((b + \bar{a}) \parallel a \parallel \bar{b}) &= \gamma(b + \bar{a}) \oplus \gamma(a) \oplus \gamma(\bar{b}) = \{\{b\}, \{\bar{a}\}\} \oplus \{\{a\}\} \oplus \{\{\bar{b}\}\} = \\
&= \{\{b, a\}, \{\bar{a}, a\}\} \oplus \{\{\bar{b}\}\} = \{\{b, a, \bar{b}\}, \{\bar{a}, a, \bar{b}\}\}
\end{aligned}$$

where there are two synchronisation pairs.

Lemma B.2. *Given a reachable revTPL configuration X , then $X \xrightarrow{\tau}$ is decidable.*

$$\begin{array}{c}
(\text{ACT}) \alpha.P \xrightarrow{\alpha}_t P \qquad (\text{SUM}_1) \frac{P \xrightarrow{\alpha}_t P'}{P + Q \xrightarrow{\alpha}_t P' + Q} \qquad (\text{SUM}_2) \frac{Q \xrightarrow{\alpha}_t Q'}{P + Q \xrightarrow{\alpha}_t P + Q'} \\
(\text{THEN}) \frac{P \xrightarrow{\alpha}_t P'}{[P](Q) \xrightarrow{\alpha}_t P'} \qquad (\text{COM}_1) \frac{P \xrightarrow{\alpha}_t P'}{P \parallel Q \xrightarrow{\alpha}_t P' \parallel Q} \qquad (\text{COM}_2) \frac{Q \xrightarrow{\alpha}_t Q'}{P \parallel Q \xrightarrow{\alpha}_t P \parallel Q'} \\
(\text{COM}_3) \frac{P \xrightarrow{\alpha}_t P' \quad Q \xrightarrow{\bar{\alpha}}_t Q'}{P \parallel Q \xrightarrow{\tau}_t P' \parallel Q'} \\
(\text{RES}) \frac{P \xrightarrow{\alpha}_t P' \quad \alpha \notin \{a, \bar{a}\}}{P \setminus a \xrightarrow{\tau}_t P' \setminus a} \qquad (\text{REC}) \frac{A \stackrel{\text{def}}{=} P \quad P \xrightarrow{\alpha}_t P'}{A \xrightarrow{\alpha}_t P'}
\end{array}$$

Figure 11: LTS of TPL: rules for actions

$$\begin{array}{c}
(\text{ACT}_2) \alpha.P \xrightarrow{\sigma}_t \alpha.P \qquad (\text{NIL}) \mathbf{0} \xrightarrow{\sigma}_t \mathbf{0} \qquad (\text{WAIT}) \sigma.P \xrightarrow{\sigma}_t P \\
(\text{SUM}_3) \frac{P \xrightarrow{\sigma}_t P' \quad Q \xrightarrow{\sigma}_t Q'}{P + Q \xrightarrow{\sigma}_t P' + Q'} \qquad (\text{THEN}_2) \frac{P \xrightarrow{\bar{\sigma}}_t}{[P](Q) \xrightarrow{\sigma}_t Q} \\
(\text{COM}_4) \frac{P \xrightarrow{\sigma}_t P' \quad Q \xrightarrow{\sigma}_t Q' \quad (P \parallel Q) \xrightarrow{\bar{\sigma}}_t}{P \parallel Q \xrightarrow{\sigma}_t P' \parallel Q'} \qquad (\text{RES}_2) \frac{P \xrightarrow{\sigma}_t P'}{P \setminus a \xrightarrow{\sigma}_t P' \setminus a} \\
(\text{REC}_2) \frac{A \stackrel{\text{def}}{=} P \quad P \xrightarrow{\sigma}_t P'}{A \xrightarrow{\sigma}_t P'}
\end{array}$$

Figure 12: LTS of TPL: rules for time

Proof. By a simple induction on the structure of X . □

APPENDIX C. ADDITIONAL PROOFS OF SECTION 4

Property 4.11 (Square Property - SP). Given a reachable revTPL configuration X and two cointitial transitions $t : X \xrightarrow{\pi_1[i]} Y$ and $s : X \xrightarrow{\pi_2[j]} Z$ with $t \mathcal{I} s$ there exist two cofinal transitions $t' : Y \xrightarrow{\pi_2[j]} W$ and $s' : Z \xrightarrow{\pi_1[i]} W$.

Proof. The proof is by case analysis on the direction of the two transitions. We distinguish three cases according to whether the two transitions are both forwards, both backwards, or one forwards and the other backwards.

t and s forwards: first we look at the case where the two actions are both communication actions. The proof is by induction on the structure of the common source configuration X . From $\mathbf{0}$ no transition is possible hence this case can never happen. For a standard

prefix, a single transition is possible, but for the choice of the key. Two transitions with the same key are not independent due to condition 4 in Definition 4.9, hence there is nothing to prove. The cases of non-standard prefixes, timeout (both standard and non standard), hiding and constants follow by inductive hypothesis. If the configuration is a parallel composition then either we apply rule [PAR] and its symmetric, or we apply rule [PAR] and rule [SYN], or two [SYN]. In the case of two applications of rule [PAR], if the same parallel component acts in both the cases, then the thesis follows from inductive hypothesis. Otherwise we have $X = X_1 \parallel X_2 \xrightarrow{\alpha_1[i]} Y_1 \parallel X_2 = Y$ and $X_1 \parallel X_2 \xrightarrow{\alpha_2[j]} X_1 \parallel Z_2 = Z$ with premises $X_1 \xrightarrow{\alpha_1[i]} Y_1$ and $X_2 \xrightarrow{\alpha_2[j]} Z_2$ (note that $i \neq j$ by case 4 of Definition 4.9). By applying rule [PAR] again we have $Y = Y_1 \parallel X_2 \xrightarrow{\alpha_2[j]} Y_1 \parallel Z_2$ and $Z = X_1 \parallel Z_1 \xrightarrow{\alpha_1[i]} Y_1 \parallel Z_2$, as desired. In the case of two applications of rule [SYN] we proceed by induction on the two components. In the case of one [PAR] and one [SYN] we proceed by induction on the component which acts in [PAR].

In the case of choice, if the two transitions concern the same component then the thesis follows by inductive hypothesis. If the two transitions concern different components then they are not independent as they are in forward communication conflict, according to condition 2 in Definition 4.9.

In the case $\pi_1 = \sigma$ and $\pi_2 = \sigma$ there is nothing to prove since then $Y = Z$ and $t = s$ by time determinism.

Finally, note that the case $\pi_1 = \alpha$ and $\pi_2 = \sigma$ is ruled out by Definition 4.9 (first clause).

t and s backwards: the case of two communication actions is similar to the previous one, noticing however that backward transitions are never in conflict according to Definition 4.9. Indeed, keys ensure that each component can take part in a single transition.

The case of two σ actions follows since time determinism holds also in the backward direction, since time actions are required on all components, and they need to have the same key.

The case of a σ action and a communication action follows from Proposition 4.10.

t forwards and s backwards: the case of communication actions is similar to the previous one: actions either are in conflict by condition 3 of Definition 4.9 or they are generated by parallel components, hence can take place also in the opposite order. Two time transitions from the same configuration are always in conflict by condition 3 of Definition 4.9, since time actions are recorded in each component. The case $\pi_1 = \alpha$ and $\pi_2 = \sigma$ (or vice versa) is analogous to the previous one. \square