How Reversibility Can Solve Traditional Questions: The Example of Hereditary History-Preserving Bisimulation

Clément Aubert
School of Computer and Cyber Sciences, Augusta University, GA, USA
http://spots.augusta.edu/caubert/
caubert@augusta.edu

Ioana Cristescu
Tarides, Paris, France
ioana@tarides.com

Abstract

Reversible computation opens up the possibility of overcoming some of the hardware's current physical limitations. It also offers theoretical insights, as it enriches multiple paradigms and models of computation, and sometimes retrospectively enlightens them. Concurrent reversible computation, for instance, offered interesting extensions to the Calculus of Communicating Systems, but was still lacking a natural and pertinent bisimulation to study processes equivalences. Our paper formulates an equivalence exploiting the two aspects of reversibility: backward moves and memory mechanisms. This bisimulation captures classical equivalences relations for denotational models of concurrency (history- and hereditary history-preserving bisimulation, (H)HPB), that were up to now only partially characterized by process algebras. This result gives an insight on the expressiveness of reversibility, as both backward moves and a memory mechanism – providing "backward determinism" – are needed to capture HHPB.

2012 ACM Subject Classification Theory of computation → Program semantics; Computing methodologies → Concurrent programming languages

Keywords and phrases Formal semantics, Process algebras and calculi, Distributed and reversible computation, Configuration structures, Reversible CCS, Bisimulation

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2020.7

Related Version A complete but preliminary research report containing all the proofs can be found at https://hal.archives-ouvertes.fr/hal-02568250.

Acknowledgements The authors would like to thank John Natale for correcting the definition of postfixing and the reviewers of an earlier version of this work and of this version for their precious comments that greatly improved the paper. We were unfortunately not able to accomodate all of their suggestions, but have tried to reflect their comments in the body of the paper.

1 Introduction

The Benefits of Reversible Computation. Future progresses in computing may heavily rely on reversibility [17]. The foreseeable limitations of conventional semiconductor technology, Landauer's principle [22] – promising low-energy consumption for reversible computers – and quantum computing [26] – intrinsically reversible and now within reach [1] – motivated a colossal push toward a better understanding of reversible computation. Those efforts have given birth to new paradigms [16], richer models of computation (e.g. for automata [20], Petri nets [14, 27], Turing machines [3]) and richer semantics, sometimes for preexisting calculi like the Calculus of Communicating Systems (CCS) [11, 28], the π-calculus [9, 10] and the higher-order π-calculus [25]. Those new perspectives sometimes additionally give in retrospect a better understanding of "traditional" (i.e., irreversible) computation, and our contribution illustrates this latter aspect.

© Clément Aubert and Ioana Cristescu;
licensed under Creative Commons License CC-BY

Summary. In brief terms, we offer a solution to an open problem on classes of equivalence of (non-reversible) concurrent processes thanks to reversibility. *Hereditary history-preserving bisimulation* (HHPB) is considered “the gold standard” for establishing equivalence classes on “true” models of concurrency [21, 36]. However, no relation expressed in syntactical terms (e.g. on CCS) was known to capture it, despite intensive efforts: previous results [2, 29] characterized HHPB on limited classes of processes, that excluded CCS terms as simple as $a.a$, $a + a$, $a \mid a$ or containing similar patterns. We prove in this paper a somewhat expected result, namely that adding mechanisms to reverse the computation and keep track of the past enables syntactical characterizations of HHPB. This result uses natural reformulations of canonical CCS bisimulations, provides a “meaningful” bisimulation for reversible calculi, and furthermore validates the mechanism we use to reverse the computation in a concurrent set-up. It also connects with previous semantics of reversible processes [2, 18] and supports categorical treatment.

Reversing Concurrent Computation. Reversible systems can backtrack and return to a previous state. Implementing reversibility often requires a mechanism to record the history of the execution. Ideally, this history should be *complete*, so that every forward step can be backtracked, and *minimal*, so that only the relevant information is saved. Concurrent programming languages have additional requirements: the history should be *distributed*, to avoid centralization, and should prevent steps that required a synchronization with other parts of the program to backtrack without undoing this synchronization. To fulfill those requirements, Reversible CCS (RCCS) [11, 12] uses *memories* attached to the threads of a process[1], and CCS with Communication Keys (CCSK) [30] “marks” each occurrence of an action. The two calculi are equi-expressive [24] and are conservative extensions over CCS, and in this paper we will use RCCS, to benefit from its previous denotational semantics [2]. Reversible calculi are also *backward deterministic*, a term introduced [6, p. 15] to denote the fact that only one piece of information – in RCCS, the identifier in a memory – is enough to undo a particular action. It should be noted that using only the labels is not enough, as it can be the case that multiple events with the same label have taken place and could be undone.

Equivalences for Denotational Models of Concurrency. A theory of concurrent computation, be it reversible or irreversible, relies not only on a syntax and a model, but also on “meaningful” behavioral equivalences. This paper defines new bisimulations on RCCS, and proves their relevance by connecting them to bisimulations on configuration structures [40], a classical denotational model for concurrency. In configuration structures, an *event* represents an execution step, and a *configuration* – a set of events that occurred – represents a state. A forward transition is then represented as moving from a configuration to one of its supersets. Backward transitions have a “built-in” representation: it suffices to move from a configuration to one of its subsets. Among the behavioral equivalences on configuration structures, some of them, like *history-* and *hereditary history-preserving bisimulation* (HPB and HHPB) [6, 32, 34, 35], use that “built-in” notion of reversibility. HPB and HHPB are usually regarded as the “canonical” [29, p. 94] or “strongest desirable true” [31, p. 2] concurrency equivalences because they preserve causality, branching, their interplay, and are

[1] In this system, the distribution of the memory is given precedence over the minimality: while there is some redundancy in the memories, they are maximally distributed, and hence every thread carries its own complete history.

the coarser (or finer, for HHPB) reasonable equivalences with these properties [37, p. 309]. HHPB also naturally equals path bisimulations [21], an elegant notion of equivalence relying on category theory, and can be captured concisely using event identifier logic [31]. However, no relation on operational semantics, e.g. on the labeled transition systems (LTS) of CCS, RCCS, or CCSK, was known to capture it on all processes.

Encoding Reversible Processes in Configuration Structures. There have been multiple attempts [2, 30] to transfer equivalences defined on denotational models, by construction suited for reversibility, into reversible process algebras. Showing that an equivalence on configuration structures corresponds to one on processes starts by defining an encoding of the latter in the former. Encodings, for RCCS [2] and CCSK [18] alike, generally consider only *reachable* reversible processes, and map them to one particular configuration in the encoding of its *origin*, the CCS "memory-less" process to which it can backtrack. We come back to the relation between this encoding and our current work in the conclusion.

Contribution. This paper improves on previous results [2, 29] by defining relations on syntactical terms that correspond to HHPB on *all* processes, and hence are proven to be "the right" bisimulation to study reversible computation[2]. This result relies on encoding the processes' memories into *identified configuration structures*, an extension to configuration structures that constitutes our second contribution.

RCCS is exploited as a syntactic tool to decide HHPB for CCS processes, by endowing them with

1. backtracking capabilities and
2. a memory mechanism.

This gives a precious insight on the expressiveness of reversibility, as we show that having only one of those tools is not enough to define relations that capture HHPB. The memories attached to a process are no longer only a syntactic layer to implement reversibility, but become essential to define and capture equivalences, thanks to the backward determinism they provide.

Recursion is not treated in this paper: its treatment amounts to unfolding processes and structures up to a certain level, and it strongly suggests that there are not much insight to gain from its development, that we reserve as a technical appendix for future work.

Related Work. The correspondences between HHPB and back-and-forth bisimulations for restricted classes of processes [2, 29] inspired some of the work presented here. This correspondence has been studied in the denotational world, on structures without auto-concurrency [6].

The insight that backward determinism is essential to capture HHPB is also used in the event identifier logic [31] and in one if its sub-systems [5], that both characterize HHPB without restricting the structures considered. As in our case, the authors exploit identifiers to eliminate constraints due to label "duplication". Our work is similar, taking place in the operational world instead of the logical one. Note that we do not introduce extra syntax constructions on the LTS, but simply use the ones provided by RCCS, that we use "as is".

[2] Of course, this claim is open for discussion [23, Conclusion], nevertheless HHPB is to the best of our knowledge the strongest form of bisimulation one can obtain on truly concurrent systems, which makes it, at the very least, a point of comparison when studying other relations.

In the operational semantics, the previous attempts to characterize HHPB wrongly focused on the backward capabilities of processes and considered the memory mechanisms only as a tool to achieve it. As a result, those bisimulations could not decide that the encoding of $(a.a) \mid b$ and $a \mid a \mid b$ were not HHPB. Instead, their characterizations of HHPB were applicable only on "non-repeating" [29] or "singly-labeled" [2] processes. By integrating the memory mechanism, the equivalence relation we introduce and consider can correctly determine that these two processes are not HHPB, as we detail in Example 27.

Finally, our approach shares similarity with causal trees – in the sense that only *part of the execution*, its "past", is encoded in a denotational representation – which were used to characterize HPB for CCS terms [13]. Capturing (H)HPB with novel techniques can also impact model-checking and decidability issues [5], but we leave this as future work.

2 Denotational and Operational Models for the Reversible CCS

We write $\subseteq$ for the set inclusion, $\mathcal{P}$ for the power set, $\setminus$ for the set difference, $f : A \to B$ (resp. $f : A \rightharpoonup B$) for the (resp. partial) function from A to B, $f\restriction_C$ for the restriction of f to $C \subseteq A$, and $f \cup \{a \mapsto b\}$ for the function defined as f on A that additionally maps $a \notin A$ to b.

2.1 Identified Configuration Structures

Labeled configuration structures [38, 39] are a classical non-interleaving model of concurrent computation – also known as "stable configuration structures" [37, Definition 5.5] or "completed stable families" [41, Section 3.2] – , that we enrich here with identifiers. We then show that they support categorical understanding and the usual operations just as well as their "un-identified" variations.

▶ **Definition 1** ((Identified) Configuration structure). *A* configuration structure $\mathcal{C}$ *is a tuple* (E, C, L, ℓ) *where E is a set of events, L is a set of* labels, *$\ell : E \to L$ is a labeling function and $C \subseteq \mathcal{P}(E)$ is a set of subsets satisfying:*

$$\forall x \in C, \forall e \in x, \exists z \in C \text{ finite}, e \in z, z \subseteq x \quad \text{(Finiteness)}$$

$$\forall x \in C, \forall d, e \in x, d \neq e \Rightarrow \exists z \in C, z \subseteq x, d \in z \iff e \notin z \quad \text{(Coincidence Freeness)}$$

$$\forall X \subseteq C, \forall x, y \in X, \exists z \in C \text{ finite}, x \cup y \subseteq z \Rightarrow \bigcup X \in C \quad \text{(Finite Completeness)}$$

$$\forall x, y \in C, x \cup y \in C \Rightarrow x \cap y \in C \quad \text{(Stability)}$$

If $\mathcal{C}$ also has a set of identifiers *I and an* identifying function *$\mathsf{m} : E \to I$ satisfying:*

$$\forall x \in C, \forall e_1, e_2 \in x, \mathsf{m}(e_1) \neq \mathsf{m}(e_2) \quad \text{(Collision Freeness)}$$

then we write $\mathcal{C} \oplus \mathsf{m}$, and say that $\mathcal{I} = (E, C, L, \ell, I, \mathsf{m})$ is an identified configuration structure*, or $\mathcal{I}$-*structure*, and call $\mathcal{C}$* the underlying configuration structure *of $\mathcal{I}$.*

We denote with $\mathbf{0}$ *both the $\mathcal{I}$-structure and its underlying configuration structure with $C = \{\emptyset\}$, and, for $x, y \in C$, we write $x \xrightarrow{e} y$ and $y \overset{e}{\rightsquigarrow} x$ if $x = y \cup \{e\}$ and $e \notin x$.*

We omit the identifiers when representing $\mathcal{I}$-structures and write the label for the event (with a subscript if multiple events shares a label).

▶ **Definition 2** (Causality, concurrency, and maximality). *For all $\mathcal{I}$, $x \in C$ and $d, e \in x$, the* causality relation on x *is given by $d <_x e$ iff $d \leqslant_x e$ and $d \neq e$, where $d \leqslant_x e$ iff for all $y \in C$ with $y \subseteq x$, we have $e \in y \Rightarrow d \in y$. The* concurrency relation on x *is given by $d \operatorname{co}_x e$ iff $\neg(d <_x e \vee e <_x d)$. Finally, x is* a maximal configuration in $\mathcal{I}$ *if $\forall y \in C$, $x = y$ or $x \not\subseteq y$.*

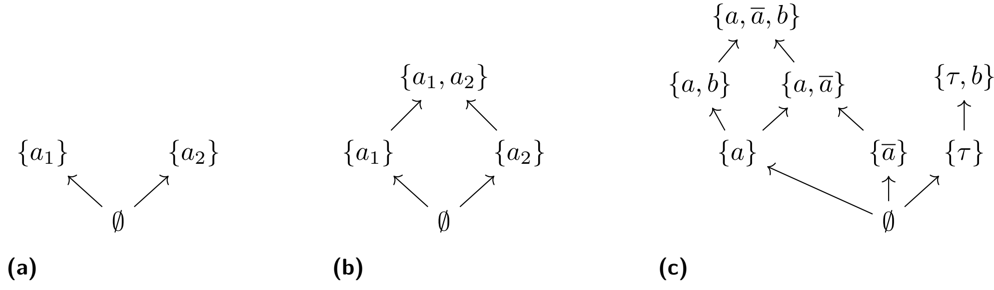


Figure 1 Examples of $\mathcal{I}$-structures.

▶ **Example 3.** Consider Fig. 1, where we let the events have distinct arbitrary identifiers: if two events with complement names as labels can happen at the same time (Fig. 1c), then they are "merged" into a single event labeled with τ, as is usual in CCS (Sect. 2.2). In Fig. 1c, $a <_{\{a,b\}} b$, $a <_{\{a,\overline{a},b\}} b$ and $\tau <_{\{\tau,b\}} b$; and in Fig. 1b, $a_1 \operatorname{co}_{\{a_1,a_2\}} a_2$. An $\mathcal{I}$-structure can have one (Fig. 1b) or multiple (Fig. 1a and 1c) maximal configurations.

Categorical point of view. We remind in Appendix A that configuration structures and "structure-preserving" functions form a category. We also prove that a similar category can be defined with $\mathcal{I}$-structures as objects and define a forgetful functor that returns the underlying configuration structure. This development supports the interest and validity of studying $\mathcal{I}$-structures, but can be omitted, except for the notion of morphisms:

▶ **Definition 4** (Morphism of $\mathcal{I}$-structure). *A morphism $f = (f_E, f_L, f_C, f_m)$ between $\mathcal{I}_1$ and $\mathcal{I}_2$ is given by $f_E : E_1 \to E_2$ such that $\ell_2(f_E(e)) = f_L(\ell_1(e))$, for $f_L : L_1 \to L_2$; $f_C : C_1 \to C_2$ defined as $f_C(x) = \{f_E(e) \mid e \in x\}$, and $f_m : I_1 \to I_2$ such that $f_m(m_1(e)) = m_2(f_E(e))$. We often write f for all the components, and write $\mathcal{I}_1 \cong \mathcal{I}_2$ if f is an isomorphism.*

Operations on $\mathcal{I}$-configurations. Operations on $\mathcal{I}$-structures are conservative extensions over their counterparts on configuration structures – forgetting about event identifiers gives back the "un-identified" definition [39] – , except for the parallel composition. The intuition here is that configuration structures encode CCS processes, and $\mathcal{I}$-structures encode memories of RCCS processes, where parallel composition has a different meaning. Examples of those operations will be given in Sect. 2.4, after introducing the calculi in Sect. 2.2 and the encodings in Sect. 2.3. Sect. C.1 gives intuitions on the correctness of those operations.

Given two sets A, B, and a symbol $\star \notin A \cup B$ denoting *undefined*, we write $C^\star = C \cup \{\star\}$ if $\star \notin C$ and define the partial product [39, Appendix A] of A and B to be

$$A \times_\star B = \{(a, \star) \mid a \in A\} \cup \{(\star, b) \mid b \in B\} \cup \{(a, b) \mid a \in A, b \in B\}$$

and the two projections to be $\pi_1 : A \times_\star B \to A^\star$ and $\pi_2 : A \times_\star B \to B^\star$.

▶ **Definition 5** (Operations on $\mathcal{I}$-structures). *Given $\mathcal{I}_i = (E_i, C_i, L_i, \ell_i, I_i, m_i)$, for $i = 1, 2$,*

The relabeling *of $\mathcal{I}_1$ along $\ell' : E_1 \to L$ is $\mathcal{I}_1[\ell'/\ell_1] = (E_1, C_1, L, \ell', I_1, m_1)$.*

The reidentifiying *of $\mathcal{I}_1$ along $m : E_1 \to I$ is $\mathcal{I}_1[m/m_1] = (E_1, C_1, L_1, \ell_1, I, m)$, provided $\mathcal{I}_1[m/m_1]$ respects the Collision Freeness condition.*

The restriction of a set of events *$A \subseteq E_1$ in $\mathcal{I}_1$ is $\mathcal{I}_1 \upharpoonright_A = (E, C, L, \ell_1\upharpoonright_E, I, m_1\upharpoonright_E)$, where $E = E_1 \backslash A$, $C = \{x \mid x \in C_1$ and $x \cap A = \emptyset\}$, $L = \{a \mid \exists e \in E_1 \backslash A, \ell_1(e) = a\}$ and $I = \{i \mid \exists e \in E_1 \backslash A, m_1(e) = i\}$.*

The restriction of a set of labels $L \subseteq L_1$ *in* $\mathcal{I}_1$ *is* $\mathcal{I}_1 \upharpoonright_L = \mathcal{I}_1 \upharpoonright_{E_1^L}$ *where* $E_1^L = \{e \in E_1 \mid \ell_1(e) \in L\}$. *We write* $\mathcal{I}_1 \upharpoonright_a$, *when the restricting set of labels* L *is the singleton* $\{a\}$.

The prefixing *of* $\mathcal{I}_1$ *by the label* a *is* $a.\mathcal{I}_1 = (E, C, L, \ell, I, m)$ *where*

- $E = E_1 \cup \{e\}$, *for* $e \notin E_1$,
- $C = \{x \mid x = \emptyset$ *or* $\exists x' \in C_1, x = x' \cup e\}$,
- $L = L_1 \cup \{a\}$,
- $\ell = \ell_1 \cup \{e \mapsto a\}$,
- $I = I_1 \cup \{i\}$, *for* $i \notin I_1$
- $m = m_1 \cup \{e \mapsto i\}$.

The postfixing *of* (a, i) *to* $\mathcal{I}_1$ *is defined if* $i \notin I_1$ *as* $\mathcal{I}_1 :: (a, i) = (E, C, L, \ell, I, m)$ *where everything is as in* $a.\mathcal{I}_1$, *except that* $C = C_1 \cup \{x \cup \{e\} \mid x \in C_1$ *is maximal and finite*$\}$.

The nondeterministic choice *of* $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *is* $\mathcal{I}_1 + \mathcal{I}_2 = (E, C, L, \ell, I, m)$, *where*

- $E = \{\{1\} \times E_1\} \cup \{\{2\} \times E_2\}$ *with* $\pi_1 : E \to \{1, 2\}$ *and* $\pi_2 : E \to E_1 \cup E_2$,
- $C = \{\{i\} \times x \mid x \in C_i\}$,
- $L = L_1 \cup L_2$,
- $\ell(e) = \ell_i(\pi_2(e))$ *for* $\pi_1(e) = i$,
- $I = I_1 \cup I_2$,
- $m(e) = m_i(\pi_2(e))$ *for* $\pi_1(e) = i$.

The product *of* $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *is* $\mathcal{I}_1 \times \mathcal{I}_2 = (E, C, L, \ell, I, m)$, *where:*

- $E = E_1 \times_\star E_2$, *with* $\pi_i : E \to E_i^\star$ *the projections of the partial product,*
- *For* $i \in \{1, 2\}$, *define the projections* $\gamma_i : \mathcal{I}_1 \times \mathcal{I}_2 \to \mathcal{I}_i$ *and the configurations* $x \in C$*:*

$$
\begin{aligned}
&\forall e \in E, \gamma_i(e) = \pi_i(e), \gamma_i(\ell_i(e)) = \ell_i(\pi_i(e)), \gamma_i(m_i(e)) = m_i(\pi_i(e))\\
&\gamma_i(x) \in C_i, \text{ with } \gamma_i(x) = \{e_i \mid \pi_i(e) = e_i \neq \star \text{ and } e \in x\}\\
&\forall e, e' \in x, \pi_1(e) = \pi_1(e') \neq \star \text{ or } \pi_2(e) = \pi_2(e') \neq \star \Rightarrow e = e'\\
&\forall e \in x, \exists z \subseteq x \text{ finite}, \gamma_i(z) \in C_i, e \in z\\
&\forall e, e' \in x, e \neq e' \Rightarrow \exists z \subseteq x, \gamma_i(z) \in C_i, e \in z \iff e' \notin z
\end{aligned}
$$

- $\ell : E \to L = L_1 \times_\star L_2$ *is* $\ell(e) = \begin{cases} (\ell_1(e_1), \star) & \text{if } \pi_2(e) = \star \\ (\star, \ell_2(e_2)) & \text{if } \pi_1(e) = \star \\ (\ell_1(e_1), \ell_2(e_2)) & \text{otherwise} \end{cases}$
- $m : E \to I = I_1 \times_\star I_2$ *is* $m(e) = \begin{cases} (m_1(\pi_1(e)), \star) & \text{if } \pi_2(e) = \star \\ (\star, m_2(\pi_2(e))) & \text{if } \pi_1(e) = \star \\ (m_1(\pi_1(e)), m_2(\pi_2(e))) & \text{otherwise} \end{cases}$

We now recall the definition of parallel composition for configuration structures, and detail the definition for $\mathcal{I}$-structures. Parallel composition consists of a combination of product, relabelling, reidentifiying for the $\mathcal{I}$-structures, and restriction. For the relabelling operation, we use a *synchronization algebra* [42] $(S, \star, \perp)$ consisting of a commutative and associative operation $\bullet$ on a set of labels $S \cup \{\star, \perp\}$, where $\{\star, \perp\} \notin S$ and such that $a \bullet \star = a$ (i.e. $\star$ is the identity element) and $a \bullet \perp = \perp$ (i.e. $\perp$ is the zero element), for all $a \in S$. To avoid repetition, below we assume given $(S, \star, \perp)$, such that $S \subseteq L_1 \cup L_2$. We give examples of synchronization algebras in Sect. 2.3.

▶ **Definition 6** (Parallel composition of configuration structures)**.** The parallel composition *of* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *is* $\mathcal{C}_1 \mid_S \mathcal{C}_2 = \big((\mathcal{C}_1 \times \mathcal{C}_2)[\ell'/\ell]\big)\upharpoonright_\perp$ *where* $\ell : E_1 \times_\star E_2 \to L_1 \cup L_2$ *is the labeling function from the product, and* $\ell' : E_1 \times_\star E_2 \to L_1 \cup L_2 \cup \{\perp\}$ *is defined as* $\ell'(e) = \ell_1(e_1) \bullet \ell_2(e_2)$.

▶ **Definition 7** (Parallel composition of $\mathcal{I}$-structures). The parallel composition *of* $\mathcal{I}_1$ *and* $\mathcal{I}_2$*, is* $\mathcal{I}_1 \mid_S \mathcal{I}_2 = (\mathcal{I}_3[m'/m_3][\ell'/\ell_3])\upharpoonright_\perp$ *where* $\mathcal{I}_3 = (E_3, C_3, L_3, \ell_3, I_3, m_3)$ *is* $\mathcal{I}_1 \times \mathcal{I}_2$*, and*

- $m' : E_3 \to I_1 \cup I_2 \cup \{\perp_k \mid k \in I_1 \times_\star I_2\}$ *is defined as follows, for* $i \neq \star$*:*

$$m'(e) = \begin{cases} i & \text{if } m_3(e) = (i,i) & \text{(Sync. or Fork)} \\ i & \text{if } m_3(e) = (i,\star) \wedge \forall e_2 \in E_2, m_2(e_2) \neq i & \text{(Extra } \star\text{. 1)} \\ i & \text{if } m_3(e) = (\star,i) \wedge \forall e_1 \in E_1, m_1(e_1) \neq i & \text{(Extra } \star\text{. 2)} \\ \perp_k & \text{otherwise, with } m_3(e) = k & \text{(Error)} \end{cases}$$

- $\ell' : E_3 \to L_1 \cup L_2 \cup \{\perp\}$ *maps* e *to* $\perp$ *if* $m'(e) = \perp_k$*, and to* $\ell_1(e_1) \bullet \ell_2(e_2)$ *otherwise.*

Parallel composition removes from the product the pairs of events that represent "non-realizable" interactions: for configuration structures, pairs of events that do not represent *possible and future synchronizations*; for $\mathcal{I}$-structures, pairs of events that do not represent *past synchronizations or forks.* Definitions 11 and 12 will detail how those operations are used to encode a process or a memory, and both types of parallel compositions will be illustrated in Examples 14 and 15.

2.2 Concurrent Communicating Calculi

Let $\mathsf{I} = \mathbb{N}$ be a set of *identifiers* and i, j range over it. Let $\mathsf{N} = \{a, b, c, \dots\}$ be a set of *names* and $\overline{\mathsf{N}} = \{\overline{a}, \overline{b}, \overline{c}, \dots\}$ its set of *co-names*. We let $\mathsf{N} \cup \overline{\mathsf{N}} \cup \{\tau\}$ be the set of *labels*, and use α (resp. λ, μ, ν) to range over them (resp. over $\mathsf{N} \cup \overline{\mathsf{N}}$). The *complement* of a name is given by a bijection $\overline{\cdot} : \mathsf{N} \to \overline{\mathsf{N}}$, whose inverse is also written $\overline{\cdot}$, and that we extend to τ, i.e. $\overline{\tau} = \tau$.

▶ **Definition 8** (RCCS Processes). *The set of* reversible processes R *is built on top of the set of CCS processes by adding memory stacks to the threads:*

$$\begin{aligned} P, Q &\coloneqq P \mid Q \;\parallel\; \textstyle\sum_{i \geqslant 0} \lambda_i.P_i \;\parallel\; P\backslash a && \text{(CCS Processes)} \\ e &\coloneqq \langle i, \lambda, P \rangle && \text{(Memory Events)} \\ m &\coloneqq \emptyset \;\parallel\; \curlyvee.m \;\parallel\; e.m && \text{(Memory Stacks)} \\ T &\coloneqq m \rhd P && \text{(Reversible Threads)} \\ R, S &\coloneqq T \;\parallel\; R \mid S \;\parallel\; R\backslash a && \text{(RCCS Processes)} \end{aligned}$$

We denote $\mathsf{I}(e)$ *(resp* $\mathsf{I}(m)$*,* $\mathsf{I}(R)$*) the set of identifiers occurring in* e *(resp.* m*,* R*), and let* $\mathrm{nm}(m) = \{\alpha \mid \alpha \in \mathsf{N}$ *or* $\overline{\alpha} \in \overline{\mathsf{N}}$ *occurs in* $m\}$ *be the set of (co-)names occurring in* m*.*

Note that the nullary case of the sum[3] gives the inactive process, denoted 0, and that the unary case gives the "usual" prefixing of a process by a label, and we write e.g. $a.P$ for $\sum_1 \lambda_i.P_i$ with $\lambda_1 = a$ and $P_1 = P$. We assume sum to be associative and often consider only its binary case, that we denote with the $+$ sign. We often forget about the trailing $\emptyset$ in the memory stack, or the inactive process 0 and write e.g. $e \rhd a \mid (b + \overline{c})$ for $e.\emptyset \rhd (a.0 \mid (b.0 + \overline{c}.0))$. We work up to the structural congruence $\equiv$ of CCS – that we suppose familiar to the reader – in CCS processes under the memory, and write e.g. $\emptyset \rhd (P_1 \mid P_2 \mid P_3)$ without parenthesis since $(P_1 \mid P_2) \mid P_3 \equiv P_1 \mid (P_2 \mid P_3)$. Finally, the only binder is restriction, and alpha-equivalence, written $=_\alpha$ and supposed familiar, equates e.g. $((a + \overline{a}) \mid b)\backslash a$ with $((c + \overline{c}) \mid b)\backslash c$.

[3] This version of sum is used for simplifying the presentation of the LTS in Fig. 2, and always written with at least one prefix to ease remembering its particular form.

$$i \notin \mathsf{I}(m) \quad \frac{}{(m \rhd \lambda.P + Q) \xrightarrow{i:\lambda} \langle i, \lambda, Q\rangle.m \rhd P} \text{ act.}$$

$$\frac{R \xrightarrow{i:\lambda} R' \qquad S \xrightarrow{i:\overline{\lambda}} S'}{R \mid S \xrightarrow{i:\tau} R' \mid S'} \text{ syn.}$$

$$i \notin \mathsf{I}(m) \quad \frac{}{\langle i, \lambda, Q\rangle.m \rhd P \stackrel{i:\lambda}{\rightsquigarrow} m \rhd (\lambda.P + Q)} \text{ act.}_*$$

$$\frac{R \stackrel{i:\lambda}{\rightsquigarrow} R' \qquad S \stackrel{i:\overline{\lambda}}{\rightsquigarrow} S'}{R \mid S \stackrel{i:\tau}{\rightsquigarrow} R' \mid S'} \text{ syn.}_*$$

$$i \notin \mathsf{I}(S) \quad \frac{R \stackrel{i:\alpha}{\Longrightarrow} R'}{R \mid S \stackrel{i:\alpha}{\Longrightarrow} R' \mid S} \text{ par.}_{\text{L}}$$

$$i \notin \mathsf{I}(S) \quad \frac{S \stackrel{i:\alpha}{\Longrightarrow} S'}{R \mid S \stackrel{i:\alpha}{\Longrightarrow} R \mid S'} \text{ par.}_{\text{R}}$$

$$\frac{R \stackrel{i:\alpha}{\Longrightarrow} R' \qquad a \notin \alpha}{R\backslash a \stackrel{i:\alpha}{\Longrightarrow} R'\backslash a} \text{ res.}$$

$$\frac{R_1 \equiv R \qquad R \stackrel{i:\alpha}{\Longrightarrow} R' \qquad R' \equiv R'_1}{R_1 \stackrel{i:\alpha}{\Longrightarrow} R'_1} \equiv$$

Figure 2 Rules of the labeled transition system (LTS).

In a memory event $\langle i, \lambda, P\rangle$, the P component represents the process that was not chosen in a non-deterministic transition, but that can be restored if the process wants to go back. The "fork" symbol $\curlyvee$ tracks when a memory stack is split between two threads.

▶ **Definition 9** (Structural equivalence [2, 11]). *Structural equivalence on* R *is the smallest equivalence relation generated by the following rules:*

$$\frac{P =_\alpha Q}{m \rhd P \equiv m \rhd Q} \qquad (\alpha\text{-Conversion})$$

$$m \rhd (P \mid Q) \equiv (\curlyvee.m \rhd P \mid \curlyvee.m \rhd Q) \qquad \text{(Distribution of Memory)}$$

$$m \rhd P\backslash a \equiv (m \rhd P)\backslash a \text{ with } a \notin \text{nm}(m) \qquad \text{(Scope of Restriction)}$$

The labeled transition system for RCCS is given by the rules of Fig. 2. We use $\stackrel{i:\alpha}{\Longrightarrow}$ for the union of $\xrightarrow{i:\alpha}$ (forward) and $\stackrel{i:\alpha}{\rightsquigarrow}$ (backward transition), and if there are indices $i_1, \dots, i_n$ and labels $\alpha_1, \dots, \alpha_n$ such that $R_1 \stackrel{i_1:\alpha_1}{\Longrightarrow} \cdots \stackrel{i_n:\alpha_n}{\Longrightarrow} R_n$, then we write $R_1 \Rrightarrow R_n$. Sect. 2.4 will provide examples of executions, but it should be noted that a process $m \rhd a.P$ is allowed to make a transition with label a and identifier $i \notin \mathsf{I}(m)$ using act. and add the event $\langle i, a, 0\rangle$ to the memory stack m if $i \notin \mathsf{I}(m)$. Conversely, a process $\langle i, a, 0\rangle.m \rhd P$ can do a backward transition using act.$_*$ with label a and identifier i and become $m \rhd a.P$. This system is a conservative extension over the LTS of CCS with prefixed sum, simply adding indices and backward transitions: in this sense, CCS can be seen as a sublanguage of RCCS, namely by identifying $\emptyset \rhd P$ with P and using only forward transitions.

▶ **Definition 10** (Reachable [2, Lemma 1]). *For all R, if there is a CCS process P such that $\emptyset \rhd P \Rrightarrow R$, we say that R is* reachable, *that P is the* unique origin *of R and write it O_R.*

An important result [11, Lemma 10] furthermore states that a forward-only[4] trace $\emptyset \rhd P \Rrightarrow R$ exists. Also, note that multiple RCCS processes can have the same origin, but that reachable RCCS processes have one unique origin (up to structural equivalence). We consider only reachable terms: unreachable terms are "dysfunctional" and their memory is considered *not coherent* [12], as they can not "rewind" back to an origin process.

[4] Traces and trace equivalences for RCCS are reminded in Appendix B: they are needed for some proofs and they are similar to their CCS's counterpart [8], but are not required to understand our results.

In RCCS, identifiers are needed to distinguish between events that have the same label. In CCS, events are considered "the same" up to labels: two events can synchrionise if they have the same label. Therefore, in the forward computation, events are free to choose a synchronization partner as long as they have the same label. Undoing a synchronization, however requires a precise pairs of events to backtrack: an event cannot change its synchronization partner when going backwards. Reversibility therefore needs to distinguish events are considered "the same" in the non-reversible setting. This constraint of using identifiers to distinguish events with the same label makes RCCS backward deterministic.

2.3 Processes and Memories as (Identified) Configuration Structures

In the definitions below, we write S for a synchronization algebra $(S, \star, \bot)$ with $S = \mathsf{N} \cup \overline{\mathsf{N}} \cup \{\tau\}$.

▶ **Definition 11** (Encoding CCS processes [42]). *Given a CCS process P, its encoding $[\![P]\!]$ as a configuration structure is built inductively:*

$$[\![\lambda.P + Q]\!] = [\![\lambda.P]\!] + [\![Q]\!] \qquad [\![P \mid Q]\!] = [\![P]\!] \mid_S [\![Q]\!] \qquad [\![P\backslash a]\!] = [\![P]\!]\upharpoonright_{\{a,\overline{a}\}}$$
$$[\![\lambda.P]\!] = \lambda.[\![P]\!] \qquad [\![0]\!] = \mathbf{0}$$

where S includes $\alpha \bullet \overline{\alpha} = \tau$ and $\alpha \bullet \beta = \bot$, if $\beta \neq \overline{\alpha}$.

▶ **Definition 12** (Encoding RCCS memories). *Given a RCCS process R, the encoding $\lceil R \rceil$ of its memory as an $\mathcal{I}$-structure is built by induction on the process and on the memory:*

$$\lceil m \rhd P \rceil = \lfloor m \rfloor \qquad \lceil R_1 \mid R_2 \rceil = \lceil R_1 \rceil \mid_S \lceil R_2 \rceil \qquad \lceil R\backslash a \rceil = \lceil R \rceil$$
$$\lfloor \langle i, \lambda, P \rangle.m \rfloor = \lfloor m \rfloor :: (\lambda, i) \qquad \lfloor \emptyset \rfloor = \mathbf{0} \qquad \lfloor \curlyvee.m \rfloor = \lfloor m \rfloor$$

where S includes $\alpha \bullet \overline{\alpha} = \tau$; $\alpha \bullet \alpha = \alpha$ and $\alpha \bullet \beta = \bot$ if $\beta \notin \{\overline{\alpha}, \alpha\}$.

Intuitively, a memory is a linear sequence of events executed by a process, which has resolved choices (i.e. in the sum or in synchronizations). The encoding of a memory is a chain of configurations. Appending a memory event corresponds to adding an event on top of the chain, and parallel composition combines two chains by fusing "partial" events resulting from a fork or a synchronization. We show examples in the following section and make this argument more formal in Lemma 17.

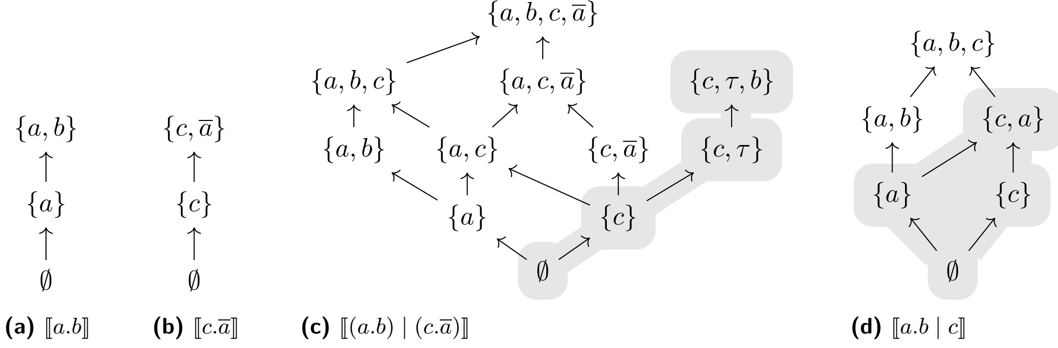
2.4 Examples

We now illustrate the execution of RCCS processes, the encoding of CCS processes and of RCCS memories, and how they relate.

▶ **Example 13** (Executing a RCCS process). An example of forward-only trace is:

$$
\begin{aligned}
\emptyset \rhd (a.b \mid c.\overline{a}) \equiv\ & (\curlyvee.\emptyset \rhd a.b) \mid (\curlyvee.\emptyset \rhd c.\overline{a}) && \text{(Distribution of Memory)}\\
\xrightarrow{1:c}\ & (\curlyvee.\emptyset \rhd a.b) \mid (\langle 1, c, 0\rangle.\curlyvee.\emptyset \rhd \overline{a}) && \text{(act.)}\\
\xrightarrow{2:\tau}\ & (\langle 2, a, 0\rangle.\curlyvee.\emptyset \rhd b) \mid (\langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle.\curlyvee.\emptyset \rhd 0) && \text{(syn.)}\\
\xrightarrow{3:b}\ & (\langle 3, b, 0\rangle.\langle 2, a, 0\rangle.\curlyvee.\emptyset \rhd 0) \mid (\langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle.\curlyvee.\emptyset \rhd 0) && \text{(act.)}
\end{aligned}
$$

Reading it from end to beginning and replacing $\xrightarrow{:}$ with $\overset{:}{\leftsquigarrow}$ gives a *backward-only* trace, that would rewind the process back to its origin. Of course, a trace can mix forward and backward transitions, as we illustrate in Example 20. The memory of this process is encoded in Example 15.



(a) $[\![a.b]\!]$ **(b)** $[\![c.\overline{a}]\!]$ **(c)** $[\![(a.b) \mid (c.\overline{a})]\!]$ **(d)** $[\![a.b \mid c]\!]$

Figure 3 $\mathcal{I}$-structures for Examples 14, 15, 16 and 20, with the CCS term encoded by their underlying configuration in caption.

▶ **Example 14** (Encoding CCS processes)**.** We can see the $\mathcal{I}$-structures from Fig. 1 as configuration structures obtained by encoding the CCS processes $a + a$, $a \mid a$, and $(a.b) \mid \overline{a}$. Similarly, we can consider the $\mathcal{I}$-structures from Fig. 3 – ignoring the grayed out parts for now – as configuration structures. The interested reader can check that the encoding of $(a.b) \mid (c.\overline{a})$ in Fig. 3c is indeed the result of applying the parallel composition of configurations structures (Definition 6) to the encoding of $a.b$ in Fig. 3a and of $c.\overline{a}$ in Fig. 3b. Lastly, Fig. 3d shows the encoding of $(a.b) \mid c$.

The parallel composition of $\mathcal{I}$-structures (Definition 7) differs slightly and is new, and hence deserves a detailed example.

▶ **Example 15** (Encoding RCCS memories)**.** The process obtained at the end of Example 13 has its memory encoded as follows:

$$\begin{aligned}
&\lceil (\langle 3, b, 0\rangle.\langle 2, a, 0\rangle. \curlyvee .\emptyset \rhd 0) \mid (\langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle. \curlyvee .\emptyset \rhd 0)\rceil \\
=&\lceil \langle 3, b, 0\rangle.\langle 2, a, 0\rangle. \curlyvee .\emptyset \rhd 0\rceil \mid \lceil \langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle. \curlyvee .\emptyset \rhd 0\rceil \\
=&\lfloor \langle 3, b, 0\rangle.\langle 2, a, 0\rangle. \curlyvee .\emptyset\rfloor \mid \lfloor \langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle. \curlyvee .\emptyset\rfloor
\end{aligned}$$

Letting $E = L = \{a, \overline{a}, b, c\}$, $\ell = \text{id}$, $I = \{1, 2, 3\}$, using $\lfloor \curlyvee .\emptyset\rfloor = \lfloor \emptyset \rfloor = \mathbf{0}$ and the postfixing:

$$\lfloor \langle 3, b, 0\rangle.\langle 2, a, 0\rangle. \curlyvee .\emptyset\rfloor = (\{a, b\}, \{\emptyset, \{a\}, \{a, b\}\}, L, \ell, I, \{a \mapsto 2, b \mapsto 3\}$$

$$\lfloor \langle 2, \overline{a}, 0\rangle.\langle 1, c, 0\rangle. \curlyvee .\emptyset\rfloor = (\{c, \overline{a}\}, \{\emptyset, \{c\}, \{c, \overline{a}\}\}, L, \ell, I, \{c \mapsto 1, \overline{a} \mapsto 2\})$$

Those are displayed in Fig. 3a and 3b, and their product (which is the first step to compute their parallel composition) gives the following sets of events and identifiers:

Event	$(a, \star)$	$(b, \star)$	$(\star, c)$	$(\star, \overline{a})$	(a, c)	$(a, \overline{a})$	(b, c)	$(b, \overline{a})$
Identifier	$(2, \star)$	$(3, \star)$	$(\star, 1)$	$(\star, 2)$	$(2, 1)$	$(2, 2)$	$(3, 1)$	$(3, 2)$

Re-identifying and re-labeling according to the definition of parallel composition gives:

Event	$(a, \star)$	$(b, \star)$	$(\star, c)$	$(\star, \overline{a})$	(a, c)	$(a, \overline{a})$	(b, c)	$(b, \overline{a})$
Re-identified	$\bot_{(2,\star)}$	3	1	$\bot_{(\star,2)}$	$\bot_{(2,1)}$	2	$\bot_{(3,1)}$	$\bot_{(3,2)}$
Re-labeled	$\bot$	b	c	$\bot$	$\bot$	τ	$\bot$	$\bot$

Indeed, if two events occur at the same time with the same identifier (Sync. or Fork), then their identifier is simply picked. Hence, $m'(a, \overline{a}) = 2$. If only one event is present in the pair, and no event on the other component have the same identifier (Extra $\star$. 1, Extra $\star$. 2),

then this event's identifier is picked. Hence, $m'(b,\star) = 3$ and $m'(\star,c) = 1$. The remaining cases get re-identified with $\bot_k$ (Error). Finally, $(b,\star)$, $(\star,c)$ and $(a,\overline{a})$ gets relabeled with b, c and τ respectively, and after restricting to the label $\bot$ we obtain the grayed out part of Fig. 3c.

Observe that in this last example, the structure underlying the encoding of the memory is just a particular "path" in the encoding of the origin. We can observe this intuition again with the following example:

▶ **Example 16** (Memory and origin). The encoding of the memory resulting from the execution $\emptyset \rhd ((a.b) \mid c) \xrightarrow{1:c} \xrightarrow{2:a} (\langle 2,a,0\rangle. \curlyvee .\emptyset \rhd b) \mid (\langle 1,c,0\rangle. \curlyvee .\emptyset \rhd 0)$ is the grayed out part in Fig. 3d, with $m(c) = 1$ and $m(a) = 2$. We name this process R_1 and come back to it in Example 20.

2.5 Operational Correspondence

Before studying bisimulations on configuration structures and processes, we prove the operational correspondence[5] between RCCS processes and the encodings of their memories in $\mathcal{I}$-structures (Lemma 19, cf. also Sect. C.2). Events in $\mathcal{I}$-structures resulting from the encoding of a process have different identifiers, and they are either causally linked or concurrent.

▶ **Lemma 17** (Memories give posets). *For all R, letting x be the maximal configuration in $\lceil R \rceil$ (Definition 2), $(\lceil R \rceil, \subseteq)$ is a partially ordered set (poset) with maximal element x.*

This is proved by induction on R and illustrated by Examples 15 and 16. However, having at most one maximal configuration does not imply that one particular event has to be "the last one" introduced. We use the following definition to make it formal.

▶ **Definition 18** (Maximal event). *An event e is* maximal *in $\mathcal{I}$ if there is no event e' such that $e <_x e'$, for x a maximal configuration of $\mathcal{I}$.*

For instance, the encoding of the memory of Example 16, pictured in Fig. 3d, has two maximal events, labeled a and c. We can now state the main result of this section:

▶ **Lemma 19** (Operational Correspondence). *For all R and S, writing $(E_R, C_R, \ell_R, I_R, m_R)$ for $\lceil R \rceil$ and similarly for S, if $R \xrightarrow{i:\alpha} S$ or $S \overset{i:\alpha}{\rightsquigarrow} R$, then there exists $e \in E_S$ maximal in $\lceil S \rceil$ with $m_S(e) = i$ s.t. $\lceil R \rceil \cong \lceil S \rceil \upharpoonright_{\{e\}}$. For all R and e a maximal event in $\lceil R \rceil$, there is a transition $R \overset{m_R(e):\ell_R(e)}{\rightsquigarrow} S$ with $\lceil S \rceil \cong \lceil R \rceil \upharpoonright_{\{e\}}$.*

For the first part, it suffices to show that the forward transition triggers the creation of a maximal event with the same identifier, and nothing else, and that this event can be "traced" in $\lceil S \rceil$. It uses intermediate lemma (Lemmas 43–45) showing how maximal events are preserved by certain operations on $\mathcal{I}$-structures. The result follows easily for backward transitions, but the last part is more involved: it requires to show that e can be mapped to a particular transition in the trace from O_R to R, and, using a notion of trace equivalence, that this particular transition can be "postponed" and done last, so that R can backtrack on it.

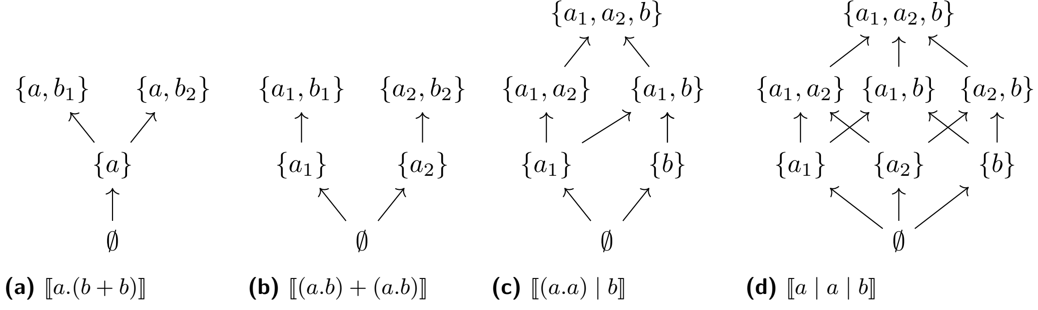
▶ **Example 20** (Forward and backward transitions). Looking back at the process of Example 16, we could further have $R_1 \overset{1:c}{\rightsquigarrow} R_2 \xrightarrow{3:b} R_3$, i.e.

$$(\langle 2,a,0\rangle. \curlyvee .\emptyset \rhd b) \mid (\langle 1,c,0\rangle. \curlyvee .\emptyset \rhd 0) \overset{1:c}{\rightsquigarrow} (\langle 2,a,0\rangle. \curlyvee .\emptyset \rhd b) \mid \curlyvee.\emptyset \rhd c)) \quad (\text{act.}_*)$$

$$\xrightarrow{3:b} (\langle 3,b,0\rangle.\langle 2,a,0\rangle. \curlyvee .\emptyset \rhd 0) \mid \curlyvee.\emptyset \rhd c)) \quad (\text{act.})$$

We can see using Fig. 3d that $\lceil R_1 \rceil \upharpoonright_c = \lceil R_2 \rceil$ and that $\lceil R_2 \rceil = \lceil R_3 \rceil \upharpoonright_b$.

[5] Similar to the operational correspondance between configuration structures and CCS processes [7, Section 3] i.e., if $P \xrightarrow{\alpha} Q$, then $[\![P]\!] = [\![Q]\!] \upharpoonright_{\{e\}}$, where e is an event in $[\![P]\!]$ such that $\ell(e) = \alpha$.



(a) $[\![a.(b+b)]\!]$ **(b)** $[\![(a.b)+(a.b)]\!]$ **(c)** $[\![(a.a) \mid b]\!]$ **(d)** $[\![a \mid a \mid b]\!]$

Figure 4 Configuration structures for Examples 23 and 27.

3 Reversible and Truly Concurrent Bisimulations Are the Same

3.1 History-Preserving Bisimulations in Configuration Structures

History-preserving bisimulation (HPB) [4, 31, 32] and hereditary history-preserving bisimulation (HHPB) [4, 6] are equivalences on configuration structures that use label- and order-preserving bijections on events. Below, assume given $\mathcal{C}_i = (E_i, C_i, L_i, \ell_i)$ for $i = 1, 2$.

▶ **Definition 21** (Label- and order-preserving functions (l&o-p)). *A function $f : x_1 \to x_2$, for $x_i \in C_i$, $i \in \{1, 2\}$ is* label-preserving *if $\ell_1(e) = \ell_2(f(e))$ for all $e \in x_1$. It is* order-preserving *if $e_1 \leqslant_{x_1} e_2 \Rightarrow f(e_1) \leqslant_{x_2} f(e_2)$, for all $e_1, e_2 \in x_1$. We write that f is* l&o-p *if it is both.*

▶ **Definition 22** (HPB and HHPB). *A relation $\mathcal{R} \subseteq C_1 \times C_2 \times (E_1 \rightharpoonup E_2)$ such that $(\emptyset, \emptyset, \emptyset) \in \mathcal{R}$, and if $(x_1, x_2, f) \in \mathcal{R}$, then f is a l&o-p bijection between x_1 and x_2 and (1) and (2) (resp. (1–4)) hold is called a* history- *(resp.* hereditary history-*)* preserving bisimulation (HPB, resp. HHPB) between $\mathcal{C}_1$ and $\mathcal{C}_2$.

$$\forall y_1, x_1 \xrightarrow{e_1} y_1 \Rightarrow \exists y_2, g, x_2 \xrightarrow{e_2} y_2, g\restriction_{x_1} = f, (y_1, y_2, g) \in \mathcal{R} \tag{1}$$
$$\forall y_2, x_2 \xrightarrow{e_2} y_2 \Rightarrow \exists y_1, g, x_1 \xrightarrow{e_1} y_1, g\restriction_{x_1} = f, (y_1, y_2, g) \in \mathcal{R} \tag{2}$$
$$\forall y_1, x_1 \overset{e_1}{\rightsquigarrow} y_1 \Rightarrow \exists y_2, g, x_2 \overset{e_2}{\rightsquigarrow} y_2, g = f\restriction_{y_1}, (y_1, y_2, g) \in \mathcal{R} \tag{3}$$
$$\forall y_2, x_2 \overset{e_2}{\rightsquigarrow} y_2 \Rightarrow \exists y_1, g, x_1 \overset{e_1}{\rightsquigarrow} y_1, g = f\restriction_{y_1}, (y_1, y_2, g) \in \mathcal{R} \tag{4}$$

We write that $\mathcal{C}_1$ and *$\mathcal{C}_2$* are HHPB (resp. HPB) *if there exists a* HHPB *(resp.* HPB*) relation between them.*

Note that HPB and HHPB are two different relations, as e.g. $(a \mid (b+c)) + (a \mid b) + ((a+c) \mid b)$ and $(a \mid (b+c)) + ((a+c) \mid b)$ have HPB but not HHPB encodings [37] .

▶ **Example 23.** The encoding of the two processes $a.(b+b)$ and $(a.b)+(a.b)$, in Fig. 4a and 4b, are HHPB: the relation can start by mapping a to a_1 or a_2, and then maps b_1 or b_2 (depending on the superset reached) to b_1 or b_2, according to the first choice made. This relation can "follow" the forward and backward movements in both structures, giving a l&o-p bijection. This example also proves that HHPB is not CCS's structural congruence.

3.2 Back-And-Forth Bisimulations in Reversible CCS

This section presents the relations we will be using, explain the restrictions on previous attempts to capture HHPB as a relation on process algebra terms [2, 29], and shows why both backward transitions *and* identifiers are needed to capture HHPB.

Below, assume given two reachable processes R_1 and R_2, and if $f : A \to B$ is such that $f(a) = b$, we write $f \setminus \{a \mapsto b\}$ for $f\restriction_{A \setminus \{a\}}$.

▶ **Definition 24** (B&F and SB&F bisimulations). *A relation $\mathcal{R} \subseteq \mathsf{R} \times \mathsf{R} \times (\mathsf{I} \rightharpoonup \mathsf{I})$ such that $(\emptyset \rhd O_{R_1}, \emptyset \rhd O_{R_2}, \emptyset) \in \mathcal{R}$ and if $(R_1, R_2, f) \in \mathcal{R}$, then f is a bijection between $\mathsf{I}(R_1)$ and $\mathsf{I}(R_2)$ and (5–8) hold is called* a back-and-forth bisimulation (B&F) between R_1 and R_2.

$$\forall S_1, R_1 \xrightarrow{i:\alpha} S_1 \Rightarrow \exists S_2, g, R_2 \xrightarrow{j:\alpha} S_2, g = f \cup \{i \mapsto j\}, (S_1, S_2, g) \in \mathcal{R} \tag{5}$$

$$\forall S_2, R_2 \xrightarrow{i:\alpha} S_2 \Rightarrow \exists S_1, g, R_1 \xrightarrow{j:\alpha} S_1, g = f \cup \{i \mapsto j\}, (S_1, S_2, g) \in \mathcal{R} \tag{6}$$

$$\forall S_1, R_1 \overset{i:\alpha}{\rightsquigarrow} S_1 \Rightarrow \exists S_2, f, R_2 \overset{j:\alpha}{\rightsquigarrow} S_2, g = f \backslash \{i \mapsto j\}, (S_1, S_2, g) \in \mathcal{R} \tag{7}$$

$$\forall S_2, R_2 \overset{i:\alpha}{\rightsquigarrow} S_2 \Rightarrow \exists S_1, g, R_1 \overset{j:\alpha}{\rightsquigarrow} S_1, g = f \backslash \{i \mapsto j\}, (S_1, S_2, g) \in \mathcal{R} \tag{8}$$

If we remove the requirements on f and g in the second part of (5–8), we call $\mathcal{R}$ a simple back-and-forth bisimulation (SB&F)*. We write that R_1* and *R_2* are B&F *(resp.* SB&F*) if there exists a* B&F *(resp.* SB&F*) relation between them.*

Other back-and-forth bisimulations have been studied for a variety of systems [15, 25], and are similar to SB&F bisimulation in the sense that they focus on backward and forward moves, but did not took the identifiers into account.

Restrictions and Previous Results.

▶ **Definition 25** (Constraints). *Given $\mathcal{C}$, if $\forall x \in C$, $\forall e_1, e_2 \in x$, $\ell(e_1) = \ell(e_2)$ implies $e_1 = e_2$, then $\mathcal{C}$ is* non-repeating *[29, Definition 3.5]. If, $\forall x \in C$, $\forall e_1, e_2 \in x$, $e_1 \operatorname{co}_x e_2$ and $\ell(e_1) = \ell(e_2)$ implies $e_1 = e_2$, then $\mathcal{C}$ is* without auto-concurrency *[37, Definition 9.5]. A process R is* non-repeating *(resp.* without auto-concurrency*) if $[\![O_R]\!]$ is.*

Those are the constraints used in showing equivalences between process algebra and configuration structures. We omitted the definition of *singly labeled* [2, Definition 26], as it does not contribute to the understanding of our results. Every non-repeating process is without auto-concurrency, but being non-repeating events and singly labeled are incomparable features. Simple processes can be repeating (e.g. $a.a$), with auto-concurency (e.g. $(a.b) \mid a$), not singly labeled (e.g. $a + a$), and the same can be said of more complex processes using similar patterns.

The first syntactical characterization of HHPB was obtained on non-repeating processes, using the "forward-reverse bisimulation" (FR) [30, Definition 6.5], which is essentially defined as B&F, with the additional requirement that $f = \mathrm{id}$. The theorem states that non-repeating CCSK processes are FR iff their encoding are HHPB [29, Theorem 5.4]. We argue that FR gives too much importance to the technical apparatus implementing reversibility. To the best of our knowledge, freshness is the only constrain imposed on identifiers in reversible works. Hence, to obtain meaningful FR equivalences, one would have to force a particular strategy for choosing the identifiers[6]. We prefer instead to impose only the freshness constraint on the identifiers, and use bijections (instead of equality on identifier) in our equivalences.

[6] For example, a possible strategy is to always pick the smallest available identifier, which then guarantees that both events labelled a in $(a.b) + (a.b)$ of Example 23 picks the same identifier, in their respective execution trace. Using this strategy we have then that $(a.b) + (a.b)$ and $a.(b + b)$ are FR equivalent. However, this strategy makes selecting an identifier a bottleneck of the system, as all the processes have to check which is the smallest available from the same "pool".

A second attempt [2] to capture HHPB used a *back-and-forth barbed congruence* on RCCS processes which was proven to correspond to HHPB on their encodings for a restricted class of processes as well, the class of singly-labeled processes.

Pinpointing the Right Reversible Bisimulation. We lift both restrictions in Corollary 31, by proving that B&F captures HHPB on *all* processes. Before doing so, let us note that even though B&F is the right notion to capture HHPB, when restricted to non-repeating processes, which are also without auto-concurrency, it does not use in a meaningful way the identifiers.

▶ **Theorem 26** (Collapsing B&F and SB&F). *If R_1 and R_2 are without auto-concurrency, then they are* B&F *iff they are* SB&F*.*

The intuition is that since two concurrent transitions sharing the same label can not be fired at the same time, the identifiers do not add any information. The proof is easy for the forward transitions, and uses an order on the transitions enforced by causality for the backward traces. In the presence of auto-concurrency, the relations differ, e.g. the process with auto-concurrency $a \mid a$ and $a.a$ are SB&F but not B&F.

▶ **Example 27** (Reversibility is *not* "just back and forth"). Observe that the bisimulation relation obtained by only considering (5–6) and ignoring the identifiers in Definition 24 is the "standard" CCS bisimulation. Hence, it could seem natural to assume that "simply adding the backwards transitions", i.e. taking (5–8) without the identifiers, giving SB&F, would be "the right" bisimulation for RCCS. Processes like $(a.a) \mid b$ and $a \mid a \mid b$ are SB&F, but their encodings, presented in Fig. 4c and 4d, are not HPB and hence not HHPB: SB&F does not account for reversibility in a satisfactory manner.

Both the bijection on identifiers *and* backward transitions are necessary to capture HHPB. Indeed, as suggested by Example 27, "simply" considering forward and backward transitions is not enough. Let us now consider the role of the bijection on identifiers a bit further. A first remark is that Theorem 26 shows that it is easy to overlook the role of identifiers when restricting the class of processes considered. Secondly, we can prove, as an immediate corollary of Theorems 29 and 30, that considering only (5–6) (*with* the identifiers) in Definition 24 gives a characterization of HPB (Corollary 49): if anything, having a bijection between identifiers – thanks to the order on events that can be deducted from it – helps getting closer to "truly concurrent" bisimulation than adding backward transitions does. However, as HPB and HHPB do not coincide, the identifiers are not enough either.

Of course, similar mechanisms could achieve similar results, but it is our hope that reversibility is fully understood as not "just" being about adding backward transitions or memories, but to use both to obtain backward determinism.

3.3 History-Preserving Bisimulations in (R)CCS

Proving our main result (Corollary 31) will use intermediate relations on processes – called HPB and HHPB as well – that use the encoding of the memories into $\mathcal{I}$-structures. Those relations are proven to correspond to (H)HPB on the encoding of the processes on one hand (Theorem 29), and the one that characterizes HHPB is proven to coincide with B&F (Theorem 30) on the other hand. The connections between formalisms and additional discussion are gathered in Sect. C.3.

▶ **Definition 28** (HPB and HHPB on RCCS). *A relation* $\mathcal{R} \subseteq \mathsf{R} \times \mathsf{R} \times (E_1 \rightharpoonup E_2)$ *such that* $(\emptyset \rhd O_{R_1}, \emptyset \rhd O_{R_2}, \emptyset) \in \mathcal{R}$ *and if* $(R_1, R_2, f) \in \mathcal{R}$ *then* f *is an isomorphism between* $\lceil R_1 \rceil$ *and* $\lceil R_2 \rceil$ *and (9) and (10) (resp. (9–12)) hold is called a* history-*(resp.* hereditary history-*)* preserving bisimulation between R_1 and R_2*.*

$$\forall S_1, R_1 \xrightarrow{i:\alpha} S_1 \Rightarrow \exists S_2, g, R_2 \xrightarrow{j:\alpha} S_2, g\restriction_{\lceil R_1 \rceil} = f, (S_1, S_2, g) \in \mathcal{R} \tag{9}$$

$$\forall S_2, R_2 \xrightarrow{i:\alpha} S_2 \Rightarrow \exists S_1, g, R_1 \xrightarrow{j:\alpha} S_1, g\restriction_{\lceil R_1 \rceil} = f, (S_1, S_2, g) \in \mathcal{R} \tag{10}$$

$$\forall S_1, R_1 \overset{i:\alpha}{\rightsquigarrow} S_1 \Rightarrow \exists S_2, f, R_2 \overset{j:\alpha}{\rightsquigarrow} S_2, g = f\restriction_{\lceil S_1 \rceil}, (S_1, S_2, g) \in \mathcal{R} \tag{11}$$

$$\forall S_2, R_2 \overset{i:\alpha}{\rightsquigarrow} S_2 \Rightarrow \exists S_1, g, R_1 \overset{j:\alpha}{\rightsquigarrow} S_1, g = f\restriction_{\lceil S_1 \rceil}, (S_1, S_2, g) \in \mathcal{R} \tag{12}$$

where we write $g\restriction_{\lceil R \rceil}$ *for the restriction of each component of* g *to* $\lceil R \rceil$*.*

We write that R_1 and R_2 are (H)HPB *if there exists a* (H)HPB *relation between them.*

Note that the definitions above reflect the definition of (H)HPB (Definition 22): the condition $(\emptyset \rhd O_{R_1}, \emptyset \rhd O_{R_2}, \emptyset) \in \mathcal{R}$ is intuitively the counterpart to the condition that $(\emptyset, \emptyset, \emptyset)$ has to be included in the relation on configuration structures. Also, f shares similarity with the l&o-p bijection, in the sense that it exists, with the identity as the component on the labels, iff there exists a l&o-p bijection between the unique maximal configurations in $\lceil R_1 \rceil$ and $\lceil R_2 \rceil$ (Lemma 46).

Relations defined on RCCS (B&F, SB&F, HPB and HHPB) straightforwardly extend to CCS, by simply stating that P_1 and P_2 are in it if $\emptyset \rhd P_1$ and $\emptyset \rhd P_2$ are too. Therefore we can state below our results in terms of CCS processes.

▶ **Theorem 29** (Equivalences). P_1 *and* P_2 *are* HHPB *(resp.* HPB*) iff* $[\![P_1]\!]$ *and* $[\![P_2]\!]$ *are.*

This result can easily be extended to *weak*-HPB and *weak*-HHPB [6, 31], which are defined by removing from (H)HPB on configuration structures (Definition 22) and on RCCS (Definition 28) the condition that f must be preserved from one step to the next one.

▶ **Theorem 30** (Equivalence (contd)). P_1 *and* P_2 *are* B&F *iff they are* HHPB*.*

Theorem 29 (resp. Theorem 30) uses our operational correspondence between RCCS processes (resp. RCCS memories) and their encodings as configuration structures [2, Lemma 6] (resp. as $\mathcal{I}$-structure (Lemma 19)) to transition between the semantic and syntactic worlds.

Our main result will come as an immediate corollary of Theorems 29 and 30.

▶ **Corollary 31** (Main result). P_1 *and* P_2 *are* B&F *iff* $[\![P_1]\!]$ *and* $[\![P_2]\!]$ *are* HHPB*.*

4 Concluding Remarks

This work offers a “definitive” answer to the question of finding a meaningful bisimulation for reversible LTS by providing relations that correspond to (H)HPB on their encodings on *all* processes. We believe this contribution is of value because:

1. This result solves a problem that was open since HHPB was defined [6], nearly 30 years ago, for which despite the use of multiple techniques, only partial results were obtained,
2. This idea in appearance simple still requires a lot of technical work, as sketched in the Appendix and detailed in `https://hal.archives-ouvertes.fr/hal-02568250`,
3. The use of reversibility (both the backtracking capability *and* the memory mechanism) is critical to characterize HHPB on syntactical terms.

This result also enforces the importance of identifiers in general and not just as part of a backtracking mechanism. Indeed, they are generally already present when concurrency is implemented, e.g. when two `Unix` threads terminate with the same signal, the parent process have the capacity of determining which process sent which signal.

As a byproduct of our result, we also proposed an encoding of RCCS memories into an "enriched" configuration structure, called identified configuration structure. This observation echoes our previous formalism [2] and similar encoding [18] in an interesting way: as mentioned in the Introduction, a reversible process R was encoded as a pair $(\llbracket O_R \rrbracket, x_R)$ made of the configuration structure encoding the origin of R, and a configuration x_R in it, called the *address* of R. The intuition was that we could "match" a partially executed process with a configuration. We can now go further by observing that $\lceil R \rceil$ is isomorphic to the $\mathcal{I}$-structure generated by x_R, which is everything "below" it. This result (Lemma 48) is used in our proof, and exemplified by Example 16: the encoding of the memory of $(\langle 2, a, 0 \rangle . \curlyvee . \emptyset \rhd b) \mid (\langle 1, c, 0 \rangle . \curlyvee . \emptyset \rhd 0)$ corresponds to the "past" of the process, whose underlying structure is grayed out in Fig. 3d, and what is left to execute – $b \mid 0$ – corresponds to the "future" of that process, and is represented by the configuration $\{c, a, b\}$ in Fig. 3d.

References

1 Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019. doi:10.1038/s41586-019-1666-5.

2 Clément Aubert and Ioana Cristescu. Contextual equivalences in configuration structures and reversibility. *Journal of Logical and Algebraic Methods in Programming*, 86(1):77–106, 2017. doi:10.1016/j.jlamp.2016.08.004.

3 Holger Bock Axelsen and Robert Glück. On reversible turing machines and their function universality. *Acta Informatica*, 53(5):509–543, 2016. doi:10.1007/s00236-015-0253-y.

4 Paolo Baldan and Silvia Crafa. Hereditary history-preserving bisimilarity: Logics and automata. In Jacques Garrigue, editor, *APLAS*, volume 8858 of *Lecture Notes in Computer Science*, pages 469–488. Springer, 2014. doi:10.1007/978-3-319-12736-1_25.

5 Paolo Baldan and Silvia Crafa. A logic for true concurrency. *Journal of the ACM*, 61(4):24, 2014. doi:10.1145/2629638.

6 Marek A. Bednarczyk. Hereditary history preserving bisimulations or what is the power of the future perfect in program logics. Technical report, Instytut Podstaw Informatyki PAN filia w Gdańsku, 1991. URL: http://www.ipipan.gda.pl/~marek/papers/historie.ps.gz.

7 Gérard Boudol and Ilaria Castellani. On the semantics of concurrency: Partial orders and transition systems. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87*, volume 249 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1987. doi:10.1007/3-540-17660-8_52.

8 Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings*, volume 354 of *Lecture Notes in Computer Science*, pages 411–427. Springer, 1988. doi:10.1007/BFb0013028.

9 Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible p-calculus. In *LICS*, pages 388–397. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.45.

10 Ioana Cristescu, Jean Krivine, and Daniele Varacca. Rigid families for CCS and the π-calculus. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, volume 9399 of *Lecture Notes in Computer Science*, pages 223–240. Springer, 2015. doi:10.1007/978-3-319-25150-9_14.

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

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

13 Philippe Darondeau and Pierpaolo Degano. Causal trees: Interleaving + causality. In Irène Guessarian, editor, *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science, La Roche Posay, France, April 23-27, 1990, Proceedings*, volume 469 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 1990. doi:10.1007/3-540-53479-2_10.

14 David de Frutos-Escrig, Maciej Koutny, and Lukasz Mikulski. Reversing steps in petri nets. In Susanna Donatelli and Stefan Haar, editors, *Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings*, volume 11522 of *Lecture Notes in Computer Science*, pages 171–191. Springer, 2019. doi:10.1007/978-3-030-21571-2_11.

15 Rocco De Nicola, Ugo Montanari, and Frits W. Vaandrager. Back and forth bisimulations. In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990. doi:10.1007/BFb0039058.

16 Michael P. Frank. Foundations of generalized reversible computing. In Iain Phillips and Hafizur Rahaman, editors, *Reversible Computation - 9th International Conference, RC 2017, Kolkata, India, July 6-7, 2017, Proceedings*, volume 10301 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2017. doi:10.1007/978-3-319-59936-6_2.

17 Michael P. Frank. Throwing computing into reverse. *IEEE Spectrum*, 54(9):32–37, September 2017. doi:10.1109/MSPEC.2017.8012237.

18 Eva Graversen, Iain Phillips, and Nobuko Yoshida. Event structure semantics of (controlled) reversible CCS. 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 102–122. Springer, 2018. doi:10.1007/978-3-319-99498-7_7.

19 Thomas Troels Hildebrandt. *Categorical Models for Concurrency: Independence, Fairness and Dataflow*. PhD thesis, BRICS, University of Aarhus, February 2000. URL: http://www.brics.dk/DS/00/1/.

20 Markus Holzer and Martin Kutrib. Reversible nondeterministic finite automata. In Iain Phillips and Hafizur Rahaman, editors, *Reversible Computation*, pages 35–51. Springer International Publishing, 2017. doi:10.1007/978-3-319-59936-6_3.

21 André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, 1996. doi:10.1006/inco.1996.0057.

22 Yonggun Jun, Momčilo Gavrilov, and John Bechhoefer. High-precision test of landauer's principle in a feedback trap. *Physical Review Letters*, 113:190601, November 2014. doi:10.1103/PhysRevLett.113.190601.

23 Vasileios Koutavas and Matthew Spaccasassi, Carloand Hennessy. Bisimulations for communicating transactions - (extended abstract). In Anca Muscholl, editor, *FoSSaCS*, volume 8412 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2014. doi:10.1007/978-3-642-54830-7_21.

24 Ivan Lanese, Doriana Medić, and Claudio Antares Mezzina. Static versus dynamic reversibility in CCS. *Acta Informatica*, November 2019. doi:10.1007/s00236-019-00346-6.

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

26 Michael Aaron Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.

27 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.

28 Iain Phillips and Irek Ulidowski. Reversing algebraic process calculi. In Luca Aceto and Anna Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lecture Notes in Computer Science*, pages 246–260. Springer, 2006. doi:10.1007/11690634_17.

29 Iain Phillips and Irek Ulidowski. Reversibility and models for concurrency. *Electronic Notes in Theoretical Computer Science*, 192(1):93–108, 2007. doi:10.1016/j.entcs.2007.08.018.

30 Iain Phillips and Irek Ulidowski. Reversing algebraic process calculi. *The Journal of Logic and Algebraic Programming*, 73(1-2):70–96, 2007. doi:10.1016/j.jlap.2006.11.002.

31 Iain Phillips and Irek Ulidowski. Event identifier logic. *Mathematical Structures in Computer Science*, 24(2), 2014. doi:10.1017/S0960129513000510.

32 Alexander Rabinovich and Boris Avraamovich Trakhtenbrot. Behavior structures and nets. *Fundamenta Informaticae*, 11(4):357–404, 1988.

33 Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. Models for concurrency: Towards a classification. *Theoretical Computer Science*, 170(1-2):297–348, 1996. doi:10.1016/S0304-3975(96)80710-9.

34 Rob J. van Glabbeek and Ursula Goltz. Refinement of actions in causality based models. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 267–300. Springer, 1989. doi:10.1007/3-540-52559-9_68.

35 Robert J. van Glabbeek. History preserving process graphs. Technical report, Stanford University, 1996. URL: http://kilby.stanford.edu/~rvg/pub/history.draft.dvi.

36 Robert J. van Glabbeek and Ursula Goltz. Equivalence notions for concurrent systems and refinement of actions (extended abstract). In Antoni Kreczmar and Grazyna Mirkowska, editors, *MFCS*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer, 1989. doi:10.1007/3-540-51486-4_71.

37 Robert J. van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001. doi:10.1007/s002360000041.

38 Robert J. van Glabbeek and Gordon D. Plotkin. Configuration structures, event structures and petri nets. *Theoretical Computer Science*, 410(41):4111–4159, 2009. doi:10.1016/j.tcs.2009.06.014.

39 Glynn Winskel. Event structure semantics for CCS and related languages. In Mogens Nielsen and Erik Meineche Schmidt, editors, *ICALP*, volume 140 of *Lecture Notes in Computer Science*, pages 561–576. Springer, 1982. doi:10.1007/BFb0012800.

40 Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986. doi:10.1007/3-540-17906-2_31.

41 Glynn Winskel. Event structures, stable families and concurrent games. Lecture notes, University of Cambridge, 2017. URL: https://www.cl.cam.ac.uk/~gw104/ecsym-notes.pdf.

42 Glynn Winskel and Mogens Nielsen. Models for concurrency. In Samson Abramsky, Dov M. Gabbay, and Thomas Stephen Edward Maibaum, editors, *Semantic Modelling*, volume 4 of *Handbook of Logic in Computer Science*, pages 1–148. Oxford University Press, 1995.

A Event Structures as Categories

Configuration structures often use the insights provided by the categorical framework [19, 33, 39, 42]. This appendix regroups the categorical treatment of (identified) configuration structures (Definition 1).

▶ **Definition 32** (Category of configuration structures). *We define $\mathbb{C}$ the category of configuration structures, where an object is a configuration structure, and a morphism $f : \mathcal{C}_1 \to \mathcal{C}_2$ is a triple (f_E, f_L, f_C) such that*

- $f_L : L_1 \to L_2$*;*
- $f_E : E_1 \to E_2$ *preserves labels:* $\ell_2(f_E(e)) = f_L(\ell_1(e))$*;*
- $f_C : C_1 \to C_2$ *is defined as* $f_C(x) = \{f_E(e) \mid e \in x\}$*.*

We write $\mathcal{C}_1 \cong \mathcal{C}_2$ if there exists an isomorphism between $\mathcal{C}_1$ and $\mathcal{C}_2$.

For simplicity, we often assume that $L_1 = L_2$, i.e., that all the configuration structures use the same set of labels, take f_L to be the identity and remove it from the notation.

▶ **Definition 33** (Category of $\mathcal{I}$-structures). *We define $\mathbb{D}$ the category of identified configuration structures, where objects are $\mathcal{I}$-structures, and a morphism $f : \mathcal{I}_1 \to \mathcal{I}_2$ is a tuple $q = (f, f_m)$ such that*

- $f = (f_E, f_C)$ *is a morphism in $\mathbb{C}$ between the underlying structures of $\mathcal{I}_1$ and $\mathcal{I}_2$,*
- $f_m : I_1 \to I_2$ *preserves identifiers:* $f_m(m_1(e)) = m_2(f_E(e))$*.*

We write $\mathcal{I}_1 \cong \mathcal{I}_2$ if there exists an isomorphism between $\mathcal{I}_1$ and $\mathcal{I}_2$.

Observe that $\mathbb{C}$ is a subcategory of $\mathbb{D}$. In both $\mathbb{C}$ and $\mathbb{D}$, composition is written $\circ$ and defined componentwise.

▶ **Lemma 34.** *Identified configuration structures and their morphisms form a category.*

Unsurprisingly, a forgetful functor and an enrichment functor can be defined between those two categories. The only assumption is that we need to suppose that every configuration structure can be endowed with a total ordering $\preceq$ on its events.

▶ **Lemma 35.** *The forgetful functor $\mathcal{F} : \mathbb{D} \to \mathbb{C}$, defined by*

- $\mathcal{F}(E, C, \ell, I, m) = (E, C, \ell)$
- $\mathcal{F}(f_E, f_C, f_m) = (f_E, f_C)$

and the enrichment functor $\mathcal{S} : \mathbb{C} \to \mathbb{D}$, defined by

- $\mathcal{S}(E, C, \ell) = (E, C, \ell, I, m)$*, where* $I = \{1, \dots, |E|\}$ *for* $|E|$ *the cardinality of* E*, and*

$$m(e) = \begin{cases} 1 & \text{if } \forall e', e \preceq e' \\ i+1 & \text{if } \exists e', e' \preceq e,\ m(e') = i \text{ and there is no } e'' \text{ s.t. } e' \preceq e'' \preceq e \end{cases}$$

- *For* $(f_E, f_C) : (E_1, C_1, \ell_1) \to (E_2, C_2, \ell_2)$*,* $\mathcal{S}(f_E, f_C) = (f_E, f_C, f_m)$*, where we let* $f_m(m_1(e)) = m_2(f_E(e_2))$*.*

are functors.

▶ Remark 36. In $\mathbb{D}$, every morphism $f = (f_E, f_L, f_C, f_m)$ from $\mathcal{I}_1$ to $\mathcal{I}_2$ is actually fully determined by f_E whenever $f_L = \text{id}$. Indeed, given $f_E : E_1 \to E_2$, then we can define for all $x \in C_1$, $f_C(x) = \{f_E(e) \mid e \in x\}$ and f_m as $f_m(m_1(e)) = m_2(f_E(e))$. We will often make the abuse of notation of writing f_E for f and reciprocally.

B Concurrency in a Trace and Trace Equivalence

We give here a quick reminder on concurrency and causality in CCS [8] and RCCS [11] traces. Aside from the convenient notation $m_{R/S}$ that represents the memory stack(s) modified by a forward transition from R to S, and of the notation $\overrightarrow{a}$ for a list of names $a_1, \cdots, a_n$, nothing new is introduced in this Section. However, the results reminded below are used in the proofs of Lemma 19 and Theorem 26.

Concurrency on events corresponds to a notion of concurrency on transitions in RCCS traces [11, Definition 7 and Lemma 8]. For this reminder we consider only concurrency and causality for forward transitions, so that CCS intuitions work equally well. We make a remark at the end about extending the concurrency to backward transitions, but it should be noted that forward and backward transitions are not mixed.

Two transitions $t_1 = R \xrightarrow{i_1:\alpha_1} R_1$ and $t_2 = R' \xrightarrow{i_2:\alpha_2} R_2$ are *composable* if $R_1 = R'$, and in this case, doing t_1 then t_2 is written as the composition $t_1; t_2$. Given n composable transitions $t_i : R_i \xrightarrow{i:\alpha_i} R_{i+1}$ and their composition $t_1; \ldots; t_n$, we say that t_i *is a direct cause of* t_k for $1 \leqslant i < k \leqslant n$ and write $t_i < t_k$ (or, for short, $i < k$) if there is a memory stack m in R_{i+1} and a memory stack m' in R_{k+1} such that $m < m'$, where the order on memory stacks is given by prefix ordering. Note that, if they exist, m and m' are unique, as memory events in reachable processes all have a different pairs (identifier, label).

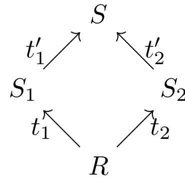
Let $R \xrightarrow{i:\alpha} S$ be a transition. If $\alpha \neq \tau$, we write $m_{R/S} = \{m\}$ where

$$R = (\cdots((R_3 \mid ((R_1 \mid (m \rhd P)) \mid R_2)\backslash\overrightarrow{b^1}) \mid R_4)\backslash\overrightarrow{b^2} \cdots \mid R_n)\backslash\overrightarrow{b^m}$$
$$S = (\cdots((R_3 \mid ((R_1 \mid \langle i, a, Q\rangle.m \rhd P) \mid R_2)\backslash\overrightarrow{b^1}) \mid R_4)\backslash\overrightarrow{b^2} \cdots \mid R_n)\backslash\overrightarrow{b^m}$$

for some R_i any of which could be missing and for some $\overrightarrow{b^j}$, possibly missing as well. If $\alpha = \tau$, then $m_{R/S}$ will contain the pair of memory stacks that has been changed by the transition. Intuitively, the notation $m_{R/S}$ is useful to extract the memory stack(s) modified by a forward transition from R to S.

Two transitions are *coinitial* if they have the same source process and *cofinal* if they have the same target process. We say that two coinitial transitions $t_1 = R \xrightarrow{i_1:\alpha_1} S_1$ and $t_2 = R \xrightarrow{i_2:\alpha_2} S_2$ are *concurrent* if $m_{R/S_1} \cap m_{R/S_2} = \emptyset$, that is, if the transitions modify disjoint memories in R.

The square lemma [11, Lemma 8] says that moreover, given two such concurrent transitions, there exists two cofinal and concurrent transitions $t'_1 = S_1 \xrightarrow{i_2:\alpha_2} S$ and $t'_2 = S_2 \xrightarrow{i_1:\alpha_1} S$. The name of the lemma comes from this picture:



Moreover, the traces $\theta_1 = t_1; t'_1$ and $\theta_2 = t_2; t'_2$ are *equivalent* [11, Definition 9]. This allows one to define *equivalence classes on transitions*: t_1 in θ_1 is equivalent to t'_2 in θ_2 if θ_1 is equivalent to θ_2 and t_1 and t'_2 have the same index. Then in the trace $t_1; t'_1$ we are now allowed to say that t_1 is concurrent to t'_1.

In a trace $t_1; t_2$ we have that t_1 is concurrent to t_2 iff t_1 is not a cause of t_2. This follows from a case analysis using the definitions of concurrency and causality. Thanks to trace equivalence, we also have that in a trace $t_1; \dots; t_n$ either t_1 is a cause of t_n or the two transitions are concurrent. Those intuitions are enough for us to carry on our development, but a complete treatment of concurrency and causality in the trace of a CCS process [8] can give better insight to the curious reader.

The definitions of concurrency for forward coinitial traces and of causality for forward traces can easily be "flipped" into definitions of concurrency for backward cofinal traces, and of causality for backward traces.

C Auxiliary Materials

In this section we introduce some intermediate definitions and lemmas that are necessary for the proofs, the details of which can be found at `https://hal.archives-ouvertes.fr/hal-02568250`.

C.1 Operations on Identified Configurations Structures (Sect. 2.1)

Our main goal here is to state that the operations of Definition 5 preserve $\mathcal{I}$-structures (Lemma 40), and to give some intuitions about them. The product and coproduct (used to define the nondeterministic choice below) have particular roles, since they have a direct representation in the categorical world (Lemma 38).

The structures we considered are *full* w.r.t. the sets of labels and identifiers, i.e. the labeling and identifying functions are surjective. This only impacts the relabelling and reidentifying operations, where we have to additionally require that ℓ' and m' are surjective.

We redefine the nondeterministic choice of Definition 5 by first defining the coproduct on $\mathcal{I}$-structures and then using relabeling and reidentifying to get rid of the extra indices in the label and the identifier of events. It is easy to check that the two definitions of nondeterministic choice are equivalent, but working with the one from Definition 5 is easier and simpler.

▶ **Definition 37.** *We redefine nondeterministic choice using coproduct as follows:*
The coproduct *of $\mathcal{I}_1$ and $\mathcal{I}_2$ is $\mathcal{I}_1 \pm \mathcal{I}_2 = (E, C, L, \ell, I, \mathsf{m})$, where*

- $E = \{\{1\} \times E_1\} \cup \{\{2\} \times E_2\}$ *with* $\pi_1 : E \to \{1, 2\}$ *and* $\pi_2 : E \to E_1 \cup E_2$,
- $C = \{\{i\} \times x \mid x \in C_i\}$,
- $L = \{\{1\} \times L_1\} \cup \{\{2\} \times L_2\}$,
- $\ell(e) = (i, \ell_i(\pi_2(e)))$ *for* $\pi_1(e) = i$,
- $I = \{\{1\} \times I_1\} \cup \{\{2\} \times I_2\}$,
- $\mathsf{m}(e) = (i, \mathsf{m}_i(\pi_2(e)))$ *for* $\pi_1(e) = i$.

and with the expected injections $\iota_i : \mathcal{I}_i \to \mathcal{I}_1 + \mathcal{I}_2$.
The nondeterministic choice *of $\mathcal{I}_1$ and $\mathcal{I}_2$ is $\mathcal{I}_1 + \mathcal{I}_2 = (\mathcal{I}_1 + \mathcal{I}_2)[\ell'/\ell][\mathsf{m}'/\mathsf{m}]$ where*

- $\ell'(e) = a$ *if* $\ell(e) = (j, a), j \in \{1, 2\}$,
- $\mathsf{m}'(e) = i$ *if* $\ell(e) = (j, i), j \in \{1, 2\}$.

▶ **Lemma 38.** *The product and coproduct of $\mathcal{I}$-structures is the product and coproduct in $\mathbb{D}$.*

In the categorical setting, the product and coproduct on labeled configuration structures can be obtained by a straightforward enrichment of un-labeled configuration structures [42, Propositions 11.2.2 and 11.2.3]. In a similar vein, we obtain the extension of those operations on identified (labeled) configuration structures directly.

The restriction of the operations of Definition 5 to configuration structures are standard [39, 40], except for postfixing. We state below that the restriction of this operation to configuration structures is correct.

▶ **Lemma 39.** *The postfixing of a label a to an event structure $\mathcal{C}_1 = (E_1, C_1, L_1, \ell_1)$, defined as $\mathcal{C}_1 :: (a) = (E, C, L, \ell)$ where*
- $E = E_1 \cup \{e\}$, *for* $e \notin E_1$,
- $C = C_1 \cup \{x \cup \{e\} \mid x \in C_1$ *is maximal and finite*$\}$,
- $L = L_1 \cup \{a\}$,
- $\ell = \ell_1 \cup \{e \mapsto a\}$,

is a configuration structure.

▶ **Lemma 40.** *The operations of Definition 5 (relabeling, reidentifiying, restriction, prefixing, postfixing, non-deterministic choice and product), coproduct, as well as the parallel composition (Definition 7) preserve $\mathcal{I}$-structures.*

C.2 Properties of Memory Encodings and Operational Correspondence

Our goal here is to give some intuition as to how to prove that that there is an operational correspondence between R and $\lceil R \rceil$ (Lemma 19) by stating intermediate lemmas (Lemmas 43–45) about the encoding of memories and their relation to maximal events. Those lemmas, in turn, requires some useful properties of memory encoding (Sect. C.2.1).

C.2.1 Properties of Memory Encodings

We assume given reachable processes R and S and we write O_R for the origin of R, and $\lceil R \rceil$ as $(E_R, C_R, \ell_R, I_R, \mathsf{m}_R)$ and similarly for S. To prove interesting properties about the encoding of memory, we first need this small technical lemma.

▶ **Lemma 41.** *For every reversible thread $m \rhd P$ of a reachable process R, and for all $i \in \mathsf{I}(m)$, i occurs once in m.*

Note that the property above holds for reversible *threads*, and not for RCCS *processes* in general: we actually *want* memory events to sometimes share the same identifiers. Indeed, two memory events need to have the same identifiers if they result from a synchronization (i.e., the application of the syn. rule of Fig. 2) or a fork (i.e., the application of the Distribution of Memory rule of structural equivalence, Definition 9).

▶ **Lemma 42** (Uniqueness of identifiers). *For all $e_1, e_2 \in E_R$, $\mathsf{m}_R(e_1) = \mathsf{m}_R(e_2)$ implies $e_1 = e_2$.*

▶ **Lemma 17** (Memories give posets). *For all R, letting x be the maximal configuration in $\lceil R \rceil$ (Definition 2), $(\lceil R \rceil, \subseteq)$ is a partially ordered set (poset) with maximal element x.*

C.2.2 Operational Correspondence

▶ **Lemma 43.** *If $R \equiv S$, then the exists an isomorphism f between $\lceil R \rceil$ and $\lceil S \rceil$, with $f_L = \mathrm{id}$ and $f_{\mathsf{m}} = \mathrm{id}$.*

▶ **Lemma 44.** *The event introduced in the postfixing of a memory event to an identified structure is maximal in the resulting identified structure.*

Furthermore, the maximality of an event can be "preserved" by parallel composition:

▶ **Lemma 45.** *For all identified structure* $\mathcal{I}_1 = (E_1, C_1, L_1, \ell_1, I_1, m_1)$ *with* $e_1 \in E_1$ *a maximal event in it, and for all identified configuration* $\mathcal{I}_2 = (E_2, C_2, L_2, \ell_2, I_2, m_2)$ *such that* $m_1(e_1) \notin I_2$, $(e_1, \star)$ *is maximal in* $\mathcal{I}_1 \mid \mathcal{I}_2$.

And then Lemmas 43–45 give all the required tools to prove the operational correspondence of Lemma 19.

C.3 Connecting Formalisms to Obtain Our Main Results

Our goal here is to give the tools and intuitions needed to prove Theorems 29 and 30, which give as an immediate corollary our main result (Corollary 31) and an interesting remark (Corollary 49). Our main task will be to identify isomorphisms of $\mathcal{I}$-structures with l&o-p functions (Lemma 46) and then to connect our previous formalism with the encoding of memories (Lemma 48). This connection coupled to the operational correspondence detailed in Sect. 2.5 makes it possible to prove our two main theorems.

We first establish a connection between isomorphisms (in category theory) and l&o-p bijections (that are used to define (H)HPB). Then, we construct a bridge between $\lfloor R \rfloor$ and our previous formalism [2].

▶ **Lemma 46.** *Letting* x_1^m *and* x_2^m *be the unique maximal configurations in* $\lceil R_1 \rceil$ *and* $\lceil R_2 \rceil$, *there is an isomorphism* f *between* $\lceil R_1 \rceil$ *and* $\lceil R_2 \rceil$ *with* $f_L = \mathrm{id}$ *iff there exists a l&o-p bijection between* x_1^m *and* x_2^m.

For the reader familiar with event structures, a configuration x defines an event structure $(x, \leqslant_x, \ell)$. The construction below mirrors the transformation from an event structure to a configuration structure [42].

▶ **Definition 47** (Generation of a $\mathcal{I}$-structure from a configuration)**.** *Given* $\mathcal{I} = (E, C, L, \ell, I, m)$, *for* $x \in C$, *the* $\mathcal{I}$-structure generated by x *is* $x{\downarrow} = (x, \{y \mid y \in C, y \subseteq x\}, \{a \mid \exists e \in x_R, \ell(e) = a\}, \ell\restriction_x, \{i \mid \exists e \in x_R, m(e) = i\}, m\restriction_x)$.

▶ **Lemma 48.** *The* $\mathcal{I}$*-structure* $\lceil R \rceil$ *is isomorphic to* $x_R{\downarrow}$, *where* $(\llbracket O_R \rrbracket, x_R)$ *is the encoding previously defined [2].*

▶ **Corollary 49.** *The relation obtained by considering only (5–6) in the definition of* B&F *(Definition 24) is equal to* HPB *on CCS terms (Definition 28).*