View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by IMT E-Theses

IMT School for Advanced Studies, Lucca

Lucca, Italy

Relative expressiveness of calculi for reversible concurrency

PhD Program in Computer Science and Engineering

XXXI Cycle

By

Doriana Medić

2018

Contents

List of Figures **v**

Abstract **viii**

1 Introduction **1**
1.1 Reversibility and Causality 1
1.2 Reversibility in CCS 4
1.3 Causality in π-calculus 7
1.4 Contributions of the thesis 12
1.5 Outline 13

2 Causality and Reversibility in CCS (background knowledge) **15**
2.1 Reversible CCS 17
2.2 CCS with Communication Keys 24

3 Encodings and Isomorphism between RCCS and CCSK **30**
3.1 Encoding CCSK into RCCS 32
3.2 Encoding RCCS into CCSK 49
3.3 Isomorphism 63
3.4 Cross-fertilisation Results 67

4 Causality notions in π-calculus (background knowledge) **72**
4.1 Causal semantics by Boreale and Sangiorgi 76
4.2 Causal semantics by Crafa, Varacca and Yoshida 80
4.3 Causal semantics for reversible π-calculus by Cristescu, Krivine and Varacca 85

5 Parametric Framework for Reversible π-Calculi **91**
5.1 Informal presentation 93
5.2 Syntax of the framework 95
5.3 Operational semantics of the framework 98
5.4 Mapping three causal semantics 105
5.4.1 Reversible semantics for the π-calculus 105
5.4.2 Boreale and Sangiorgi causal semantics 107
5.4.3 Crafa, Varacca and Yoshida causal semantics 110
5.5 Properties of the framework 112
5.5.1 π-calculus correspondence 113
5.5.2 Causal-consistency of the reversible framework . . 114
5.6 Correspondence with Boreale and Sangiorgi's semantics . 123
5.7 Causal Bisimulation 130

6 Conclusion and Future work **133**

A Appendix **135**

B Appendix **147**

References **153**

List of Figures

1 CCS syntax 16
2 CCS semantics 18
3 RCCS syntax 18
4 RCCS semantics 21
5 RCCS structural laws 21
6 CCSK syntax 25
7 CCSK forward semantics 27
8 CCSK backward semantics 28

9 Encoding of CCSK into RCCS 33
10 CCSK with irreversible actions syntax 70
11 CCSK with irreversible actions forward semantics 71

12 π-calculus syntax 74
13 π-calculus semantics 75
14 Causal semantics rules 79
15 Event structure semantics for π-calculus 81
16 The event structures representing processes P_1, P_2 and P_3 . 82
17 $R\pi$ syntax 86
18 $R\pi$ forward semantics 88
19 $R\pi$ structural laws 89

20 Syntax of the framework 96
21 Common rules for all instances of the framework. 100
22 Parametric rules 101

23 Backward rules. 104

Publications

1. Doriana Medic and Claudio Antares Mezzina, "Static VS Dynamic Reversibility in CCS," in *Reversible Computation - 8th International Conference, 2016, Bologna, Italy*, volume 9720 of Lecture Notes in Computer Science, pages 36-51. Springer, 2016.
2. Doriana Medic and Claudio Antares Mezzina, "Towards Parametric Causal Semantics in pi-calculus," *Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic*, volume 9720 of CEUR Workshop Proceedings, pages 121-125, 2017.
3. Doriana Medic, Claudio Antares Mezzina, Iain Phillips and Nobuko Yoshida, "A Parametric Framework for Reversible Pi-Calculi," *EXPRESS/SOS*, volume 276 of EPTCS, pages 87-103, 2018
4. Ivan Lanese, Doriana Medic and Claudio Antares Mezzina, "Static VS Dynamic Reversibility in CCS," submitted to Acta Informatica

Abstract

The main motivations for studying reversible computing comes from the promise that reversible computation (and circuits) would lead to more energy efficient computers. Besides circuits, nowadays, reversibility is studied in many other domains. This thesis studies the expressiveness of the causal-consistent reversibility (a well-known notion of reversibility for concurrent systems) in CCS and π-calculus.

First, we show that by means of encodings, LTSs of Reversible CCS (introduced by Danos and Krivine) and CCS with Communications Keys (introduced by Phillips and Ulidowski) are isomorphic up to some structural transformations of processes. An explanation of this result is the existence of one causality notion in CCS.

In π-calculus, two forms of dependences between the actions give rise to different causal semantics. The main difference is how the parallel extrusion of the same name is treated. We consider three approaches to parallel extrusion problem represented with causal semantics introduced by Boreale et al; Crafa et al; and Cristescu et al. To study them, we devise a framework for reversible π-calculi, parametric with respect to the data structure used to keep track of information about a name extrusions. We show that reversibility induced by our framework is causally-consistent and prove causal correspondence between the semantics given by Boreale et al, and the corresponding instance of the framework.

Chapter 1

Introduction

This thesis studies the relations between different models for reversible concurrent computations and provides a common framework in which different models can be compared. These models are based on process algebras. A process algebra can be seen as formal language endowed with very few basic primitives necessary to describe a system behaviour from an abstract point of view.

Starting from the calculus of communicating systems (CCS), a widely adopted process algebra, we investigate the expressive power of two well-known reversible extensions of CCS. We will investigate whether these two extensions are equivalent, what will allow us to have results transfer from one model to the other, and vice versa.

By moving to a more complex model featuring name creation (π-calculus) and name passing (e.g., names can be sent as value) there are several proposals on what its causal semantics should be. We then propose a reversible framework able to map some of these proposals, and we highlight the differences among them.

1.1 Reversibility and Causality

The main motivation for studying reversible computing dates back to the 60's when Landauer [48] discovered that only irreversible computation

generates heat. The aim was to reduce heat dissipation in computing machinery, and thus achieve higher density and speed. Since then there has been a huge debate whether the Landaure's principle is practically valid [2; 12; 29].

Today, reversibility is studied in different domains: biological and chemical modelling [18; 25; 46; 68; 69], since many biochemical reactions are by nature reversible; program debugging [13; 28; 30; 54; 81] and testing [75], where during debugging time program can go back to the state where certain conditions are met; quantum computing [33], since most operations are naturally reversible; thermodynamical physics [5]; and reliable systems, where many models such as transactions [35], system-recovery schemes [27] and checkpoint-rollback protocols [47], rely on some forms of undo. Distributed reversible actions can represent the structural blocks for different transactional models [24; 49] and recovery schemes.

An important property of reversible system is ability to detect the *last action* of a computation. In a sequential setting, reversibility is well understood [42], since there is only one order of the computation, executions of the backward steps start from the very last action that system performed. For instance, if the system performed actions abc, then the backward steps start from the very last action c followed by b and a. The application of reversibility in sequential setting is given with Janus [56; 78], the first reversible-structured programming language; and RFUN [79], the reversible functional programming language. The main interest is in reducing the garbage size [80] and executing the reversible simulation without generating the garbage data [4]. A reversible object-oriented language Joule^R with the implementation based on Janus, is given in [74].

Recent work on reversing imperative programming languages is given in [38; 39] where a state-saving approach to reverse an imperative program is used. One of the main challenges was modification of the initially defined method to support parallelism.

Reversibility in a concurrent setting is more complex. For instance, if system $a.b \mid c$ performs forward actions a, c and b, one could ask what is a correct backward computation? One of the solutions could be executing

backward actions in the opposite order of the forward steps, i.e. reversing b, followed by c and then a in the end. Is it possible to reverse actions with ordering c, b, a? In the concurrent setting, notion of the last action is not clearly defined. A suitable definition of what is the last action in a concurrent system is given by notion of causally-consistent reversibility. It is presented by Danos and Krivine for reversible CCS [23]. Causally-consistent reversibility connects causality and reversibility of a concurrent system through the definition of the last action: an action can be reversed (considered as the last one), only if all its consequences have been reversed. Aside from causal-consistent reversibility, other works consider different notions of reversibility: out-of-causal order [64; 66; 68; 69; 76] and backtracking [39; 64]. They are mostly used to model different biochemical systems.

In this thesis, we focus on causal-consistent reversibility for concurrent systems described via process algebras [11], in particular: Calculus of Communicating Systems (CCS) [60] and π-calculus [73].

Beside in the setting of process algebra, there are some recent works in which reversibility is studied in terms of Petri Nets [7; 8; 9; 64]. Reversibility in [7; 9] is achieved by adding reversed versions of the chosen transitions in a Petri Net. Another approach to reversibility in Petri Net is given in [64] and it is obtained by distinguishing between two notions of tokens: notion of base tokens, which represent the basic entities occurring in the system identified with unique names; and bonds tokens, new type of token created by transitions. By reversing a transition, all tokens are shifted from the outgoing edges to the incoming places of the transition and bonds created by transition are interrupted. The authors give semantics that capture three different types of reversibility: backtracking, causal reversibility and out-of-causal-order reversibility. The encoding of reversible Petri nets into a coloured Petri nets [41] is given in [8].

A process algebra can be seen as a tool for the high-level description of interactions, communications, and synchronisations between a collection of independent processes. It also provides algebraic laws that allow process descriptions to be manipulated and analysed, and permit formal reasoning about equivalences between processes (e.g. using bisimulation).

To make a concurrent calculus reversible, we need to answer on two challenges at the same time: how to keep track of past actions and how to ensure that causality relation between the actions will be respected.

In CCS, there is only one notion of causality, so-called *structural causality*, which is induced by the prefixing operator '.' and by synchronisations. Despite this intuition, there exist two different approaches to reversibility in CCS, namely Reversible CCS (RCCS), introduced by Danos and Krivine [23] and CCS with communication keys (CCSK), introduced by Phillips and Ulidowski [70]. In Section 1.2 we will review these two approaches and all the works that have sprang from them. Differently from CCS, in π-calculus exist two kinds of dependences between the actions: *subject* (structural) and the *object* dependency. Different interpretation of the object dependency give rise to different causal semantics for π-calculus which we recall through a common example in Section 1.3.

1.2 Reversibility in CCS

The first approach to reversibility in CCS is introduced in [23], where the main idea is to attach memories to every CCS process. History information of the process is kept into the memories organised as stacks of events, with the very last event that process did, on the top of the stack. The reversible process is of the form $m \triangleright P$, where m is a memory and P is a CCS process. Memory shared by two processes in parallel, needs to be duplicated and attached to both processes separately, through the structural rule. Two obtained reversible processes evolve independently. In Section 2.1, we give more details about RCCS.

In [70], Phillips and Ulidowski presented a general method for reversing process calculi, given in a SOS [71] format. The reversible extension is obtained through two main points: communication keys attached to every action used to uniquely identify actions; and every operator of the calculus is made static. The second reversible CCS, called CCS with communication keys (CCSK) is obtained by applying method given in [70] on CCS. Executed actions are not discarded, but annotated with the communication keys. Thanks to that and static operators, history information

is spread along the structure of the process, hence there is no need for external memories. In Section 2.2, we give more details about CCSK.

Two approaches mentioned above, have evolved independently for more than 10 years, giving rise to many extensions and results, for RCCS [3; 20; 22; 24; 25; 43] and CCSK [34; 45; 46; 65; 67; 68; 77]. Some of them finds their application in dependable systems or biology and chemistry modelling, while some are used for studying different models of reversibility.

Distinction between reversible and irreversible (commit) actions in RCCS is used to describe transactions [24], where commit of the successful transaction, is modelled by irreversible action. A method for the verification of distributed systems which uses an algorithm of relative causal compression was proposed in [43]. The processes one wants to verify must use a generic backtracking mechanism. Additionally, the original RCCS [23] is improved by simplifying the handling of memories and by making the splitting through the parallel composition commutative. The approach to reversibility introduced in RCCS is shifted and adapted to π-calculus [20].

Different biochemical systems can be modelled by extensions of RCCS and CCSK. In particular, transcription of a protein in DNA, can be described by CCS-R [25], calculus based on RCCS. The asynchronous RCCS [23] can be encoded into a eversible structures given in [18], used to model DNA three-domains stand.

An extension of the CCSK with the execution control mechanism which allows one to control the direction of the computation, is given in [68]. Process X, controlled by Y is represented as $X\langle Y\rangle$. In this way the computation direction of the process X is dictated by the controller Y. For instance, if $X = a.b.b$ and $Y = b.\underline{a}$, then $X\langle Y\rangle$ executes until the first action b in X is executed (from the left in textual representation) and then reverse until it does not undo the action a. The new obtained calculus is used to model the ERK signalling pathway, which delivers signals from the membrane of a cell to its nucleus. This was the first process calculus for so called out-of-causal order reversible computation (computation in which reversible steps are not respecting causality). While in [68] control

mechanism is external to the process it controls and can have global scope, in [44; 45; 46], authors introduced reversible process calculus CCB with a prefixing operator inspired by the model of covalent bonding. The new prefixing operator provides a local control of the direction of computation. The new prefix is of the form $(s; b).P$, where s is a sequence of actions or executed actions and b is a weak action. Action b can happen only provided that all actions in s were executed. Performing b then forces undoing one of the past actions in s, let say it is action $a \in s$. Then bond is promoted from the weak b to the the strong a, after what, weak action b can bond again. This type of locally controlled reversibility is shown to be suitably to model hydration of formaldehyde in water into methanediol.

A subcalculus of RCCS (RCCS without auto-concurrency, auto-conflict and recursion) can be modelled in configuration structures [3]. The authors studied relation induced on configuration structures by the barbed back-and-forth congruence. The interpretation of RCCS in rigid families is given in [22]. In rigid families causality and concurrency are derived from a partial order (precedence) which is specific to each run of a process. They are suitable for reversible calculi because the inclusion between sets dictates the allowed forward and backward transitions. The authors proposed a certain correction criteria which their model enjoys.

While RCCS can be represented with rigid families, CCSK can be seen as an example of the more general concept, given in [65], where prime graphs are used as an equivalent model to labelled prime event structures. In [66; 67] authors investigate conflict and causation for event structures with reversibility. To gain on expressiveness, regarding to the reversible form of prime event structure (RPES), they obtain more general model, reversible asymmetric event structures (RAES). Continuing work on reversible event structures, in [77] is proposed how to model reversible concurrent computation in two different reversible event structures: one defined with causation and prevention relations (Reversible Asymmetric Event Structures) and other by enabling relation (Reversible Event Structures).

Reversibility in CCSK is controlled by adding rollback operator and Roll-CCSK is obtained in [34]. Authors defined a category of reversible

bundle event structures and used its causal subcategory to define CCSK semantics. Additionally they defined a category of reversible extended bundle event structures and model Roll-CCSK with it.

Differently from CCS, in π-calculus exist two kinds of dependences between the actions: *subject* (structural) and the *object* dependency. Different interpretation of the object dependency give rise to different causal semantics for π-calculus, revised in Section 1.3.

1.3 Causality in π-calculus

In the π-calculus, differently form the CCS, possibility of creating a new channel names and treating channels as sent values, is allowed. In this way π-calculus can describe concurrent systems whose network configuration may change during the computation. As name creation is enabled, computing without forgetting becomes difficult because of the variable substitutions and more in general value passing.

Causal dependencies between the actions in π-calculus were studied by Boreale and Sangiorgi in [14], where two forms of causal dependencies were separated : *subject* (structural) and the *object* dependency. As in CCS, subject dependency in the π-calculus is determined by the nesting of the prefixes and by synchronisation. For instance, in the process $\overline{b}a.c(x)$, action $c(x)$ structurally depends on the output action $\overline{b}a$. Object dependency is generated by extruding (or opening) a name. To illustrate it, consider the process $\nu a(\overline{b}a \mid a(x))$, the output action $\overline{b}a$ extrudes bound name a and in that way enable execution of the action $a(x)$. Therefore, we say that action $a(x)$ is object dependent on the action $\overline{b}a$. The interesting problem that raises up from the object causality is a parallel extrusion of the same name. For instance, if we consider the process $\nu a(\overline{b}a \mid \overline{c}a \mid a(x))$, discussion is about which of the extrusions will cause the action $a(x)$. Depending on the solution to this problem we have different causal semantics for the π-calculus. Here we give the intuition of them trough the common example of the parallel extrusion problem, where actions are executed in the following order $\overline{b}a, \overline{c}a, a(x)$.

- *Boreale and Sangiorgi causal semantics.* The compositional causal semantics for standard (forward only) π-calculus is given in [14]. To keep track about the structural dependences, authors introduced a causal term $\mathtt{K} :: A$, where $\mathtt{K}$ represents a set of the structural causes of the process A. Object dependency is defined on the run of the process and solution for the example of the parallel extrusion of the same name is that the first extrusion of the name a will cause both actions $\overline{c}a$ and $a(x)$. In general, first extrusion of a bound name, cause every action using that name in any position (subject or the object one). More details are given in Section 4.1.

- *A data-flow analysis of the π-calculus.* In [40], authors represented the semantics of the CCS and π-calculus with the data-flow charts. Intuitively, processes can be seen as the set of the functions, while names are represented as points in the domain. For instance, domain for CCS is domain of vertical natural numbers $\mathcal{N}^v$, while for π-calculus, due to dynamical change of the process, domain is more complex. The authors do not distinguish between two types of causality (processes $\nu a(\overline{b}a \mid a(x))$ and $\nu a(\overline{b}a.a(x))$, translated to the chart semantics are equated). Regarding to our common example, we have that the final order of the action would be that the first action $\overline{b}a$ triggers both following actions using the name a.

- *Montanari and Pistore semantics for the π-calculus.* π-calculus semantics represented in the terms of graph rewriting, is given in [61]. Processes are represented as graphs, while evolution is given with graph rewriting rules. The global part of the graph keeps track of the free names, while the local part models the rest. In our common example, we have that the extrusions of the name can be executed in parallel and the input action $a(x)$ pick one of them as its cause.

- *A petri nets semantics for the π-calculus.* In the [17] authors represents the semantics of the π-calculus with the Petri Nets equipped with the inhibitor arcs. Intuitively, the state is represented by the number of tokens in each place where every token is decorated with its past (i.e. event which produce it). The inhibitors are used to prevent

transitions with a bounded name. An event can happen if every inhibiting place is empty (there is no token in it). Causal dependences are defined as: an event is caused by the set of events that produce the tokens that it uses. Regarding to our example, we have that the first extrusion $\overline{b}a$ will cause other two action using name a (regardless of the position in the label).

- *Degano and Priami semnatics.* In [26], authors studied order between the actions in π-calculus and distinguish between causality relation and precedence. Following [14], they considered two types of dependences: structural and link (object) one, while precedence is defined as temporal dependency. The union between causality and precedence is defined as *enabling* relation. For instance in the common example $\nu a(\overline{b}a \mid \overline{c}a \mid a(x))$, where actions are executed in the following order $\xrightarrow{\overline{b}a}\xrightarrow{\overline{c}a}\xrightarrow{a(x)}$, we have that the output $\overline{b}a$ has: a temporal precedence over the output $\overline{c}a$; and it is object cause of the action $a(x)$.

  The order between the actions is recorded by specifying which component of the process is performing a move. To uniquely identify the actions, authors used labels of the form $\vartheta\mu$ and $\vartheta\langle\|_0\ \vartheta_0\mu_0, \|_1\ \vartheta_1\mu_1\rangle$ where $\vartheta \in \{\|_0, \|_1\}^*$ represents the position of the subprocess performing the action, and $\mu_0 = b(x)$ iff μ_1 is either $\overline{b}a$ or $\overline{b}\langle\nu a\rangle$, or vice versa. To record that the left or the right component in the process is executing, tag $\|_0$ or $\|_1$ is used. In [26], authors stated that by abstracting away from the technique to keep track about the causality, the order of the cations induced by their semantics coincides with the one in [14; 17].

- *Crafa, Varacca and Yoshida semantics.* A compositional event structure semantics for the forward π-calculus is introduced in [19]. The event structure semantics of the π-calculus process is represented as a pair $(\mathbf{E}, \mathbf{X})$, where $\mathbf{E}$ is the prime event structure and $\mathbf{X}$ is a set of bound names. Structural causality is tracked by causal relation of $\mathbf{E}$, while the object one is defined with the notion of permitted configuration. In the common example, we would have that extrusions of the name

a are executed concurrently, while the action $a(x)$ depends on one of the extruders (configurations $\{\overline{b}a, a(x)\}$ and $\{\overline{c}a, a(x)\}$ are both permitted), but it is not necessary to record on which one. We recall this notion of causality in Section 4.2.

- *Cristescu, Krivine and Varacca causal semantics.* A compositional semantics for the reversible π-calculus is introduced in [20]. The approach to reversibility is similar as in [23] for Reversible CCS, but adapted to the π-calculus. Past information is recorded into the memory attached to every process. A reversible process has the form of $m \triangleright P$, where m is a memory and P is π-calculus process. Dependences induced by extruding of the name are captured by context causality. In the case of common example, actions $\overline{b}a$ and $\overline{c}a$ are executed concurrently, while the input action $a(x)$ can choose one of the extruders for its cause, exactly which one is determined by the context. The chosen cause will be saved in the memory of the process together with the action and its identifier. Detailed description of the reversible semantics is given in Section 4.3.

- *Reversible π-calculus represented with the rigid families.* In [21] authors introduced a rigid families as a causal model for the π-calculus. Rigid families are the model based on the configuration structures with a partial order relation defined on events. It is a temporal relation between events in the run of a process. The authors showed weak form of the correspondence between reversible process introduced in [20] and their translation in rigid families. There is a slight difference between these two reversible semantics, since in the rigid families temporal order is explicit as well as the causal one. Considering our common example, causality is similar as the one in [20].

- *A stable non-interleaving operational semantics for the π-calculus.* In [37], authors give the early causal semantics for the π-calculus defined with transition:

$$(\bar{H}, \underline{H}) \vdash P \xrightarrow[u,D]{\alpha} (\bar{H}', \underline{H'}) \vdash P'$$

where u is a location label, identifying the location of the prefixes in the term involved into a transition; histories D recording the past extruding events that non-output names in α depend on, and $\bar{H}, \underline{H}$ are extrusion histories which in the parallel composition record the location on the prefixes that extrude, receive a name, respectively. Link or object causality is captured by histories. Considering our example we have that input action $a(x)$ depends on the one of the extruders. Hence event representing the action $a(x)$ is spilt based on the extrusion history.

Additionally, authors argued about difference in the link causality, depending on wether early or late semantics is used. To illustrate this, consider the π-calculus process $(\nu a)(\nu b)(\overline{c}a \mid \overline{d}b \mid a(x))$. Taking into account early semantics, after extrusions of the names a and b, the input action could be labelled with $a(b)$ (extruded name b is received over the channel a). In this case, bout names in the input action are extruded in the past, hence, action $a(b)$ depends on both extruders. If late semantics is considered, then this situation does not exist, since name is substituted in the continuation of the input prefix, after the synchronisation.

- *Reversible high-order π-calculus ($\rho\pi$).* Reversible asynchronous higher-order π-calculus [51; 52] relies on simple name tags for identifying threads and on process terms called memories which are in charge of storing a single computation. Its operational semantics given in terms of reduction semantic and it preserves the classical structural congruence laws of the π-calculus. Reversibility in $\rho\pi$ is causally consistent and the causal information used to support reversibility in $\rho\pi$ is consistent with the one used in [14]. Since $\rho\pi$ is given in terms of reduction semantic, we will not consider the common example.

1.4 Contributions of the thesis

This thesis contributes to the better understanding of the relations between the different approaches to reversibility in CCS [50; 58] and provides a framework able to express different causality notions for π-calculus [59].

Given a two reversible extensions of the CCS, namely RCCS and CCSK, a natural question arises: are these two reversible calculi equivalent? This question is relevant, since the two approaches have evolved independently for more than 10 years. By proving equivalence between these two methods for reversing CCS, we show that from an abstract point of view there exists only one reversible CCS.

We give a positive answer to the question above, showing that the labeled transition systems of CCSK and RCCS (up to a few structural transformations on processes) are isomorphic. The consequence of isomorphism is that they can be equated by any behavioral equivalence, such as bisimilarity or trace equivalence. The proof of isomorphism relies on two encodings, one from CCSK to RCCS and the other in the opposite direction. We proved correctness of our encoding in terms of operational correspondence. Additionally, we showed that no uniform encoding can exist since reachable RCCS processes are not closed under parallel composition.

In the second part of the thesis, we focus on the causal semantics for π-calculus. Different interpretations of the object causality lead to different causal semantics for the π-calculus. We mainly consider three different causal semantics: causal semantics introduced by Boreale and Sangiorgi [14], notions of causality introduced by Crafa, Varacca and Yoshida [19] (both semantics are given for forward only π-calculus) and causal semantics for reversible π-calculus, given by Cristescu, Krivine and Varacca [20].

To study different approaches to causality in π-calculus, represented by three causal semantics mentioned above, in the context of reversible computation, we need a model which can express different notions of causality. We observed that object causality depends on the extruded names and on how information about the extruders is stored. Therefore,

we devise a framework for reversible π-calculi, parametric with respect to the data structure that stores information about the extrusions of a name. As a reversing method, we extended the approach given in CCSK [70], limited to calculi defined with GSOS [1] inference rules (e.g., CCS, CSP) to work with π-calculus. Different approaches to causality in π-calculus, can be obtained by using a different data structures. Additionally, the framework adds reversibility to the semantics that were defined only for the forward computations. Hence, it permits different orders of the causally-consistent backwards steps. We proved that reversibility introduced by the framework is causally-consistent and show causal correspondence between causal semantic given in [14] and corresponding instance of the framework. Additionally, we give the idea of the causal bisimulation that can be defined on the framework.

1.5 Outline

The rest of the thesis is structured in the following way:

Chapter 2 recalls two reversible extensions of CCS, namely RCCS and CCSK and points out the syntactical differences between them.

Chapter 3 introduces two encodings, one of CCSK into RCCS (Section 3.1) and another one of RCCS into CCSK (Section 3.2) and prove their correctness. Then we show an isomorphism between label transition systems of CCSK and RCCS (up to some structural transformation on processes) relying on two encodings (Section 3.3). Finally, thanks to our results, we show how is possible to bring some properties from one calculus to the other one (Section 3.4).

Chapter 4 recalls syntax and semantics of the π-calculus, and its three well-known causal extensions: causal semantics introduced by Boreale and Sangiorgi, notions of causality introduced by Crafa, Varacca and Yoshida (both semantics are given for forward only π-calculus) and causal semantics for reversible π-calculus, given by Cristescu, Krivine and Varacca.

Chapter 5 introduces a syntax (Section 5.2) and operational semantics (Section 5.3) of a framework for reversible π-calculi. After that in Section 5.4, we define the data structures with operations on them and map observed causal semantics into the framework. We proved that reversibility in our framework is well defined by proving standard properties for reversible calculus, Loop Lemma, Square Lemma and causal consistency (Section 5.5). Finally, in Section 5.6, we show causal correspondence between causal semantic given in [14] and corresponding instance of the framework. In the end of this Chapter (Section 5.7), we present the idea about causal bisimulation defined on the framework.

Chapter 6 concludes our work and discuss perspectives of future work.

Chapter 2

Causality and Reversibility in CCS (background knowledge)

In this Chapter we shall give the background knowledge necessary to obtain results of the thesis stated in Chapter 3.

We recall two reversible extensions of CCS [60] (Calculus of Communicating Systems), namely RCCS [23; 43] and CCSK [70]. In RCCS, reversibility is achieved by adding a memory to every CCS process in which all the necessary information are stored. CCSK is a reversible calculus obtained as a result of applying a general approach [70] to a CCS. There is a slight difference in the CCS considered by these two reversible calculi. As addition to that, neither of underlying CCSs do not take into account recursion operator. However, following the guidance in [43; 70] it can be easily added to both calculi. The CCS considered by RCCS is defined with a guarded choice and it features silent actions, while CCS considered by CCSK allows unguarded choice but it features no τ actions. In our research, we focused on the reversibility mechanisms, therefore, to uniform technical handling and to have a finer correspondence, we considered as the base of both reversible calculi, CCS used in [23; 43]. It is defined with guarded choice and it features τ actions. It also do not take into account recursion.

Let $\mathcal{A}$ be a set of actions ranged over by a, and $\overline{\mathcal{A}}$ the corresponding

$$\text{(CCS Processes)} \quad P, Q ::= \mathbf{0} \mid \sum_{i \in I} \alpha_i.P_i \mid (P \mid Q) \mid P \backslash a$$

$$\text{(Actions)} \quad \alpha ::= a \mid \overline{a} \mid \tau$$

Figure 1: CCS syntax

set of co-actions, that is $\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$. We let μ, λ and their decorated versions to range over the set $\mathtt{Act} = \mathcal{A} \cup \overline{\mathcal{A}}$, while we let α, β and their decorated versions to range over the set $\mathtt{Act}_\tau = \mathtt{Act} \cup \{\tau\}$, where $\tau \notin \mathtt{Act}$ is the *silent* action.

The syntax of the CCS is presented in Figure 1. $\mathbf{0}$ represents the idle process. A prefix (or action) can be an input a, an output $\overline{a}$ or the silent action τ. The choice operator is represented with $\sum_{i \in I} \alpha_i.P_i$, where P_i stands for the continuation to be executed after the actions α_i. We write $\alpha_j.P_j + Q$ where $Q = \sum_{i \in I \setminus \{j\}} \alpha_i.P_i$ to highlight a specific alternative, otherwise we write $\alpha.P$ for unary choice. We assume that if $I = \emptyset$, then $\sum_{i \in I} \alpha_i.P_i = \mathbf{0}$. A parallel composition of processes P and Q is represented with $P \mid Q$, while $P \backslash a$ denotes that name a is restricted to the process P (scope of the name a is process P). The set of all CCS processes is denoted with $\mathcal{P}$.

In the CCS, restriction is the only binder. We write $\mathtt{n}(P)$ for the set of names of a process P, and, respectively, $\mathtt{fn}(P)$ and $\mathtt{bn}(P)$ for the sets of free and bound names. Set of bound names is formally defined as:

Definition 1 *Set of a bound names of the given process P,written as $\mathtt{bn}(P)$, is inductively defined on the structure of the π-calculus process as:*

$$\mathtt{bn}(P \mid Q) = \mathtt{bn}(P) \mid \mathtt{bn}(Q) \qquad \mathtt{bn}(\sum_{i \in I} \alpha_i.P_i) = \bigcup_{i \in I} \mathtt{bn}(P_i)$$

$$\mathtt{bn}(\nu a(P)) = \{a\} \cup \mathtt{bn}(P)$$

Names which are not bound, are free.

Now we give a definition for the LTS, used to obtain the operational semantics of the calculi.

Definition 2 (LTS) *A* labeled transition system *(LTS) is a triple* $\langle \mathcal{P}, \mathcal{L}, \rightarrow \rangle$ *where* $\mathcal{P}$ *is a set of states,* $\mathcal{L}$ *a set of labels, and* $\rightarrow \subseteq \mathcal{P} \times \mathcal{L} \times \mathcal{P}$ *a transition relation. We write* $P \xrightarrow{l} P'$ *when* $\langle P, l, P' \rangle \in \rightarrow$.

Using the definition of LTS (Definition 2), we give a definition of the operational semantics for CCS.

Definition 3 (CCS Semantics) *The operational semantics of CCS is defined as LTS* $(\mathcal{P}, \rightarrow, \mathtt{Act}_\tau)$*, where* $\mathcal{P}$ *is the set of CCS processes and* $\mathtt{Act}_\tau$ *is the set of labels. Transition relations* $\rightarrow$ *is the smallest relations induced by the rules in Figure 1.*

The semantics of CCS is given in Figure 2. Process $\alpha.P$ can execute the action α, by applying the rule ACT. Performed action is discarded and computations continues with the process P. Rule SUM handles the choice operator. Rule PAR-L (PAR-R) allows left (right) component in the paralel composition to perform action α. Two processes can communicate through the rule SYN. Rule RES allows restricted process to execute the action α, with condition that α is not a restricted name. In order to have a better intuition about CCS semantics we give the following example.

Example 1 Let us consider process $P = a.b + c \mid d$ and execution of the action a:

$$a.b + c \mid d \xrightarrow{a} b \mid d$$

Note that action a is consumed and discarded together with the alternative process c in the choice operator. The resulting process can $b \mid d$ continue further execution.

2.1 Reversible CCS

In this Section we recall Reversible CCS [23; 43]. The first approach to reversibility in CCS was introduced in [23]. It differs from the presentation in [43] in a way how memories and splitting of the memory through the parallel composition are handled. In this work we will use the representation of RCCS appeared in [43] since it simplifies our technical development. However, as remarked in [43], the two presentations are

$$\text{(ACT)}\quad \alpha.P \xrightarrow{\alpha} P \qquad \text{(SUM)}\quad \frac{P_j \xrightarrow{\alpha} P'_j \quad j \in I}{\sum_{i \in I} P_i \xrightarrow{\alpha} P'_j} \qquad \text{(PAR-L)}\quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{(PAR-R)}\quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{(SYN)}\quad \frac{P \xrightarrow{a} P' \qquad Q \xrightarrow{\bar{a}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{(RES)}\quad \frac{P \xrightarrow{\alpha} P' \qquad \alpha \notin \{a, \bar{a}\}}{P \backslash a \xrightarrow{\alpha} P' \backslash a}$$

Figure 2: CCS semantics

$$\begin{array}{rl} \text{(RCCS Processes)} & R, S ::= m \triangleright P \mid \ (R \mid S) \ \mid R \backslash a \\ \text{(Memories)} & m ::= \langle\rangle \mid \langle k, \alpha, Q \rangle \cdot m \mid \langle\uparrow\rangle \cdot m \end{array}$$

Figure 3: RCCS syntax

conceptually the same. More details about the differences between two representations shall be given in Remark 1.

In RCCS, reversibility is achieved by adding a memory to each process. All necessary informations about the past actions are stored in the memories, organised as a stacks (memory event representing the last action that was executed is placed on the top of the stack).

Let $\mathcal{K}$ be an infinitive denumerable set of action keys, ranged over $k, h, w, \ldots$ We assume that sets $\mathtt{Act}$ and $\mathtt{Act}_\tau$ have the same definition as for CCS and that, $\mathcal{K} \cap \mathtt{Act} = \emptyset$. Let $\mathtt{ActK} = \mathtt{Act} \times \mathcal{K}$ be the set of pairs formed by an action μ and a key k. In the same way we define $\mathtt{ActK}_\tau = \mathtt{Act}_\tau \times \mathcal{K}$.

The syntax of RCCS is presented in Figure 3. Reversible processes are built on the top of CCS by adding a memory to every CCS process. *Monitored* process is a term of a form $m \triangleright P$, where m is a memory and P is a CCS process. A parallel composition of two reversible processes R and S is given by $R \mid S$, while $R \backslash a$ represents the fact that name a is

restricted in process R. *Memories* are denoted with m and represented as stack of events. The memory event representing the last action that reversible process performed is on the top of the stack (on the left in the textual representation). The empty memory is represented with $\langle\rangle$. A memory event which keeps track of the action that monitored process did is denoted with $\langle k, \alpha, Q\rangle$, where elements of a triple are action α, identified with the *key* k and alternative process Q, discarded by execution of the action α. Event $\langle\uparrow\rangle$ symbolise the fact that process was split in two parallel subprocesses. We denote by $\mathtt{Mem}$ the set of all memories. We let e to range over events and write $e \cdot m$ for the memory obtained by pushing event e on the top of the given memory m. We shall introduce the operator @ to emphasise the case when two memories m_1 and m_2 are given[1]. We write $m_1 @ m_2$ for the memory obtained by pushing all the elements in m_1 on top of m_2 (preserving their order). We will assume that the operator $\cdot$ has precedence over @. The @ operator is formally defined as follows.

Definition 4 *We define $m_1 @ m_2$ by structural induction on m_1:*

$$\begin{array}{rcl} \langle\rangle @ m_2 & = & m_2 \\ (e \cdot m_1) @ m_2 & = & e \cdot (m_1 @ m_2) \end{array}$$

To make a direct connection between keys in two reversible calculi, we define a function computing the set of keys in a given memory or in a given process. This function is not presented in [43].

Definition 5 *The set of keys in a memory m, or in a RCCS process R, written respectively $\mathtt{key}(m)$ and $\mathtt{key}(R)$, is inductively defined as follows:*

$$\begin{array}{rcl} \mathtt{key}(\langle\uparrow\rangle \cdot m) = \mathtt{key}(m) & \qquad & \mathtt{key}(\langle k, \alpha, Q\rangle \cdot m) = \{k\} \cup \mathtt{key}(m) \\ \mathtt{key}(\langle\rangle) = \emptyset & & \mathtt{key}(m \triangleright P) = \mathtt{key}(m) \\ \mathtt{key}(R \backslash a) = \mathtt{key}(R) & & \mathtt{key}(R \mid S) = \mathtt{key}(R) \cup \mathtt{key}(S) \end{array}$$

As in CCS, the only binder in RCCS is the restriction. The functions $\mathtt{n}$, $\mathtt{fn}$ and $\mathtt{bn}$ can be extended to RCCS processes and memories. Based on Definition 2, we can give a definition of the operational semantics for RCCS.

[1]Operator @ is not appearing in [23; 43]

Definition 6 (RCCS Semantics) *The operational semantics of RCCS is defined as a pair of LTSs on the same set of states and set of labels: a forward LTS* $(\mathcal{P}_R, \rightarrow, \mathtt{ActK}_\tau)$ *and a backward LTS* $(\mathcal{P}_R, \rightsquigarrow, \mathtt{ActK}_\tau)$*, where* $\mathcal{P}_R$ *is the set of RCCS processes. We define* $\rightleftharpoons = \rightarrow \cup \rightsquigarrow$*. Transition relations* $\rightarrow$ *and* $\rightsquigarrow$ *are the smallest relations induced by the rules in Figure 4 (left and right columns, respectively). Both relations exploit the structural congruence relation* $\equiv$*, which is the smallest congruence on RCCS processes containing the rules in Figure 5.*

The RCCS semantics is given in Figure 4. Let us comment on the forward rules (left column). Rule R-ACT generates a fresh new key k which is bound to the executing action α. A new memory event $\langle k, \alpha, Q\rangle$, is created and stored on the top of the memory stack. Rule R-PAR-L allows left component of the parallel composition to execute a forward action α, under the condition that $k \notin \mathtt{key}(S)$ (the key k of the executing action is not used by the parallel process). This check guarantees uniqueness of the keys. Similar for the rule R-PAR-R. The communication between two parallel processes can be done through the rule R-SYN. To do so, processes have to match both the action α and the key k. Rule R-RES allows a restricted process to move under the condition that the executed action is not the restricted name. Rule R-EQUIV allows one to exploit structural congruence.

Structural rules are given in Figure 5. Rule SPLIT allows a monitored process with a toplevel parallel composition to split by duplicating the memory and annotating both memories with $\langle\uparrow\rangle$. Obtained monitored processes can continue their computing independently. We give the following example to illustrate semantics of RCCS.

Example 2 Let us consider CCS process $P = a.b + c \mid d$ from Example 1. The initial RCCS process in this case is $R = \langle\rangle \triangleright (a.b + c \mid d)$. To execute action a, process R, needs first to split parallel composition into two separated components by applying structural rule SPLIT:

$$\langle\rangle \triangleright (a.b + c \mid d) \equiv \langle\uparrow\rangle \cdot \langle\rangle \triangleright a.b + c \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d$$

Now, action a identified by k can be executed. Then we have:

$$\langle\uparrow\rangle \cdot \langle\rangle \triangleright a.b + c \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d \xrightarrow{k,a} \langle k, a, c\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright b \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright d = R'$$

(R-ACT)

$$\frac{k \notin \mathtt{key}(m)}{m \triangleright \alpha.P + Q \xrightarrow{k,\alpha} \langle k, \alpha, Q\rangle \cdot m \triangleright P}$$

(R-ACT$^\bullet$)

$$\frac{k \notin \mathtt{key}(m)}{\langle k, \alpha, Q\rangle \cdot m \triangleright P \overset{k,\alpha}{\rightsquigarrow} m \triangleright \alpha.P + Q}$$

(R-PAR-L)

$$\frac{R \xrightarrow{k,\alpha} R' \qquad k \notin \mathtt{key}(S)}{R \mid S \xrightarrow{k,\alpha} R' \mid S}$$

(R-PAR-L$^\bullet$)

$$\frac{R \overset{k,\alpha}{\rightsquigarrow} R' \qquad k \notin \mathtt{key}(S)}{R \mid S \overset{k,\alpha}{\rightsquigarrow} R' \mid S}$$

(R-PAR-R)

$$\frac{S \xrightarrow{k,\alpha} S' \qquad k \notin \mathtt{key}(R)}{R \mid S \xrightarrow{k,\alpha} R \mid S'}$$

(R-PAR-R$^\bullet$)

$$\frac{S \overset{k,\alpha}{\rightsquigarrow} S' \qquad k \notin \mathtt{key}(R)}{R \mid S \overset{k,\alpha}{\rightsquigarrow} R \mid S'}$$

(R-SYN)

$$\frac{R \xrightarrow{k,\alpha} R' \qquad S \xrightarrow{k,\bar{\alpha}} S'}{R \mid S \xrightarrow{k,\tau} R' \mid S'}$$

(R-SYN$^\bullet$)

$$\frac{R \overset{k,\alpha}{\rightsquigarrow} R' \qquad S \overset{k,\bar{\alpha}}{\rightsquigarrow} S'}{R \mid S \overset{k,\tau}{\rightsquigarrow} R' \mid S'}$$

(R-RES)

$$\frac{R \xrightarrow{k,\alpha} R' \qquad \alpha \notin \{a, \bar{a}\}}{R\backslash a \xrightarrow{k,\alpha} R'\backslash a}$$

(R-RES$^\bullet$)

$$\frac{R \overset{k,\alpha}{\rightsquigarrow} R' \qquad \alpha \notin \{a, \bar{a}\}}{R\backslash a \overset{k,\alpha}{\rightsquigarrow} R'\backslash a}$$

(R-EQUIV)

$$\frac{R \equiv R' \qquad R' \xrightarrow{k,\alpha} S' \qquad S' \equiv S}{R \xrightarrow{k,\alpha} S}$$

(R-EQUIV$^\bullet$)

$$\frac{R \equiv R' \qquad R' \overset{k,\alpha}{\rightsquigarrow} S' \qquad S' \equiv S}{R \overset{k,\alpha}{\rightsquigarrow} S}$$

Figure 4: RCCS semantics

(SPLIT)	$m \triangleright (P \mid Q) \equiv \langle\uparrow\rangle \cdot m \triangleright P \mid \langle\uparrow\rangle \cdot m \triangleright Q$	
(RES)	$m \triangleright P\backslash a \equiv (m \triangleright P)\backslash a$	if $a \notin \mathtt{fn}(m)$
(α-CONV)	$R \equiv S$	if $R =_\alpha S$

Figure 5: RCCS structural laws

We can notice that executed action, together with identifier and alternative process c is saved in the memory of the process. Differently from the Example 1, resulting process R' is able to execute the backward action, by restoring the information from the memory $\langle k, a, c\rangle$.

If restricted name is not in the memory of a process, restriction can be pushed outside of monitored processes, by applying the rule RES. Structural rule α-CONV[2] is not appearing neither in [23] nor [43]. We add it to allow one to exploit α-conversion. Why it is necessary, we can see in the following example.

Example 3 Le us consider a monitored process R:

$$R = \langle k, \overline{a}, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (b\backslash a)$$

According to RCCS semantics, the process R is a deadlocked process and it cannot execute the forward action b. The only possibility for R to execute forward is to apply structural rule RES to move restriction outside of the monitored process. This cannot be done because of the side condition of the rule. This problem can be solved by α-conversion rule α-CONV:

$$\langle k, \overline{a}, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (b\backslash a) \equiv \langle k, \overline{a}, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (b\backslash c) \equiv (\langle k, \overline{a}, \mathbf{0}\rangle \cdot \langle\rangle \triangleright b)\backslash c \xrightarrow{w,b}$$

As we can notice, process $(\langle k, \overline{a}, \mathbf{0}\rangle \cdot \langle\rangle \triangleright b)\backslash c$ can execute forward action b.

Backward rules are given in Figure 4 (right column). All the rules are symmetric to the forward one. We shall comment just the interesting ones. Monitored process can revert its last action through the rule R-ACT$^\bullet$. It is done by taking the event from the top of the memory stack and using its information to bring process back the state before the action was executed. Rule R-PAR-L$^\bullet$ (R-PAR-R$^\bullet$) allows left (right) component in the parallel composition to execute a backward action, under the condition that the key k of the action does not belong to monitored processes in parallel. With this condition, partial undo of some synchronisations is avoided. In the following example, we explain why this check is necessary and give an intuition how semantics works.

[2] As far as we know, in RCCS literature, α-conversion appears first time in [3]

Example 4 Let us consider the RCCS process $R = m_1 \triangleright a.P \mid m_2 \triangleright \overline{a}.Q$. There are two possibilities for process R to move forward:

- process R can perform two independent actions: forward input a and output $\overline{a}$, identified with keys k and k_1, respectively:

$$m_1 \triangleright a.P \mid m_2 \triangleright \overline{a}.Q \xrightarrow{k,a} \langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright P \mid m_2 \triangleright Q$$
$$\xrightarrow{k_1,\overline{a}} \langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright P \mid \langle k_1, \overline{a}, \mathbf{0}\rangle \cdot m_2 \triangleright Q = R_1$$

  From the process R_1 to restore the process R, backward actions can be executed in any order, for example:

$$\langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright P \mid \langle k_1, \overline{a}, \mathbf{0}\rangle \cdot m_2 \triangleright Q \overset{k,a}{\rightsquigarrow} m_1 \triangleright P \mid \langle k_1, \overline{a}, \mathbf{0}\rangle \cdot m_2 \triangleright Q$$
$$\overset{k_1,\overline{a}}{\rightsquigarrow} m_1 \triangleright a.P \mid m_2 \triangleright \overline{a}.Q = R$$

- process R can perform a synchronisation on the channel a.

$$m_1 \triangleright a.P \mid m_2 \triangleright \overline{a}.Q \xrightarrow{k,\tau} \langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright P \mid \langle k, \overline{a}, \mathbf{0}\rangle \cdot m_2 \triangleright Q = R_2$$

  Executing the backward step from process R_2, From process R_2, it is not possible that just one part of the parallel composition undoes its action with key k. For instance, the right component cannot undo its action unless the left component undoes its action too. It is forbidden by the side condition in rule R-PAR-R$^\bullet$.

  Suppose that we avoid the side condition of the rule R-PAR-R$^\bullet$, then we would have the following transition (not normally allowed):

$$\langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright P \mid \langle k, \overline{a}, \mathbf{0}\rangle \cdot m_2 \triangleright Q \overset{k,\overline{a}}{\rightsquigarrow} \langle k, a, \mathbf{0}\rangle \cdot m_1 \triangleright a.P \mid m_2 \triangleright Q = R_3$$

  Process R_3 is obtained if the left parallel component in R synchronises with the environment and process R_2 is obtained when the two parallel components in R synchronise with each other. We can notice that process R_3 is a legal future of the process R, but not the legal past state of R_2.

Remark 1 In the RCCS introduced in [23], memories were defined as:

$$\text{(Memories)} \quad m ::= \langle\rangle \mid \langle m_*, \alpha, Q\rangle \cdot m \mid \langle 1\rangle \cdot m \mid \langle 2\rangle \cdot m$$

where m_* can be either a memory m (if process synchronised with another monitored process which memory was m); or $*$ if process synchronised with its environment. Events $\langle 1 \rangle$ and $\langle 2 \rangle$ indicate that monitored process with a toplevel parallel operator was split along a parallel composition according to the following structural rule:

$$m \triangleright (P \mid Q) \equiv \langle 1 \rangle \cdot m \triangleright P \mid \langle 2 \rangle \cdot m \triangleright Q$$

As we can notice, in the presentation of RCCS [43] that we use, m_* is replaced with the unique action keys and event $\langle \uparrow \rangle$ is used instead of $\langle 1 \rangle$ and $\langle 2 \rangle$. The improvements of the calculus, stated above, simplified the handling of memories and made the splitting through the parallel composition commutative. For instance, given two CCS processes $P = a.b \mid c$ and $Q = c \mid a.b$, after execution of the action a in RCCS, we obtain processes $P' = \langle *, a, \mathbf{0} \rangle \cdot \langle 1 \rangle \cdot \langle \rangle \triangleright b \mid \langle 2 \rangle \cdot \langle \rangle \triangleright c$ and $Q' = \langle 1 \rangle \cdot \langle \rangle \triangleright c \mid \langle *, a, \mathbf{0} \rangle \cdot \langle 2 \rangle \cdot \langle \rangle \triangleright b$. We can notice that action b is not identified with the same past memory in the processes P' and Q', since $\langle *, a, \mathbf{0} \rangle \cdot \langle 1 \rangle \cdot \langle \rangle \triangleright b \neq \langle *, a, \mathbf{0} \rangle \cdot \langle 2 \rangle \cdot \langle \rangle \triangleright b$.

In the following we will consider only reachable processes, as standard in the RCCS literature.

Definition 7 (Reachable Process) *A RCCS process R is* initial *if it has the form $\langle \rangle \triangleright P$. A RCCS process R is* reachable *if it can be derived from an initial process by using the rules in Figures 4 and 5.*

As stated before, one of the main property of reversible calculi, is the Loop Lemma [23, Lemma 6], which states that any reduction step can be undone.

Lemma 1 (RCCS Loop Lemma) *For each pair of reachable RCCS processes R and R', $R \xrightarrow{k,\alpha} R'$ iff $R' \overset{k,\alpha}{\rightsquigarrow} R$.*

The proof follows from the symmetry between the forward and the backward rules.

2.2 CCS with Communication Keys

In this Section we present a reversible CCS obtained by applying the general approach introduced in [70] on the CCS version underlying RCCS

$$\text{(CCSK Processes)} \quad X, Y ::= \mathbf{0} \mid \sum_{i \in I} \pi_i.X_i \mid (X \mid Y) \mid X \backslash a$$

$$\text{(CCSK Prefixes)} \quad \pi ::= \alpha \mid \alpha[k]$$

Figure 6: CCSK syntax

(defined in the beginning of this Chapter). As we already mentioned, two CCSs differs in the definition of the choice operator and silent prefixes. The obtained calculus, we will call CCSK since it is very similar to the one in [70].

In [70] the authors present a reversible extensions for process calculi defined by using Structural Operational Semantics (SOS) [71] rules. The used format is a subset of a path format [6] consisting of dynamic and static rules. The difference between this two types of rules is that in the dynamic one, operator is destroyed by the transition, while in the static one operator remains after the application of the rule. For instance, in CCS, dynamic rule would be the rule SUM, while the static one would be the rule PAR-L. To illustrate this, consider a CCS process $P = P_1 \mid P_2$. By applying the rule for parallel, we obtain the precess $P' = P_1' \mid P_2$, where the operator for parallel is present in the process P' as well. On the other hand, if we consider CCS process $Q = Q_1 + Q_2$, after applying the rule SUM, where action is executed on the process Q_1, we obtain $Q' = Q_1'$. In the process Q', we can notice that operator $+$ disappeared.

The main ideas of the approach in [70] are to make all the operators of the CCS static and to use a communication keys to identify the actions. To make operators static, performed actions shall not be discarded, but annotated as having been performed and saved in the structure of the process, representing its past. Result of having all the operators static is that there is no loss of information, therefore, there is no need to use external memory.

The syntax of CCSK is presented in Figure 6. The difference regarding the CCS is that prefix α might be annotated with identifier k, called key, written as $\alpha[k]$. We will use the same set of keys $\mathcal{K}$ as in RCCS since they

have the same role in both calculi.

As for RCCS, the only binder in CCSK is restriction. Functions $\mathtt{n}$, $\mathtt{fn}$ and $\mathtt{bn}$ can be extend from CCS to CCSK in expected way. We also define a function computing the set of keys in a given CCSK process.

Definition 8 *The set of keys of a process X, written $\mathtt{key}(X)$, is inductively defined as follows:*

$$\mathtt{key}(\alpha.P) = \mathtt{key}(\mathbf{0}) = \emptyset \qquad \mathtt{key}(\alpha[k].X) = \{k\} \cup \mathtt{key}(X)$$

$$\mathtt{key}(X \mid Y) = \mathtt{key}(X) \cup \mathtt{key}(Y) \qquad \mathtt{key}(\sum_{i \in I} \pi_i.X_i) = \bigcup_{i \in I} \mathtt{key}(\pi_i.X_i)$$

$$\mathtt{key}(X \backslash a) = \mathtt{key}(X)$$

Definition 9 *A key k is fresh in a process X, written $\mathtt{fresh}(k, X)$ if $k \notin \mathtt{key}(X)$.*

The notion of a *standard* process is defined bellow.

Definition 10 (Standard process) *A CCSK process X is standard, written $\mathtt{std}(X)$, if $\mathtt{key}(X) = \emptyset$.*

As we can notice, standard CCSK process coincide with CCS process.

Notation 1 In the following, we may drop the predicate $\mathtt{std}(X)$ and instead of it use simply P to highlight that CCSK process X is actually CCS process (process P do not contain keys).

We can now define the operational semantics of CCSK based on Definition 2.

Definition 11 (CCSK Semantics) *The operational semantics of CCSK is defined as a pair of LTSs on the same set of states and set of labels: a forward LTS $(\mathcal{P}_K, \rightarrow, \mathtt{ActK}_\tau)$ and a backward LTS $(\mathcal{P}_K, \rightsquigarrow, \mathtt{ActK}_\tau)$, where $\mathcal{P}_K$ is the set of CCSK processes. We define $\rightleftharpoons = \rightarrow \cup \rightsquigarrow$. Transition relations $\rightarrow$ and $\rightsquigarrow$ are the smallest relations induced by the rules in Figures 7 and 8, respectively and closed under α-conversion $=_\alpha$.*

Note that only structural congruence rule included in the semantics of CCSK is α-conversion.

(K-ACT1)

$$\alpha.P \xrightarrow{\alpha[k]} \alpha[k].P$$

(K-ACT2)

$$\frac{X \xrightarrow{\beta[h]} X' \qquad k \neq h}{\alpha[k].X \xrightarrow{\beta[h]} \alpha[k].X'}$$

(K-RES)

$$\frac{X \xrightarrow{\alpha[k]} X' \qquad \alpha \notin \{a, \bar{a}\}}{X \backslash a \xrightarrow{\alpha[k]} X' \backslash a}$$

(K-SUM)

$$\frac{X_j \xrightarrow{\alpha[k]} X'_j \ \forall i \neq j. X_i = X'_i \wedge \mathtt{std}(X_i)}{\sum_{i \in I} X_i \xrightarrow{\alpha[k]} \sum_{i \in I} X'_i}$$

(K-PAR-L)

$$\frac{X \xrightarrow{\alpha[k]} X' \qquad \mathtt{fresh}(k, Y)}{X \mid Y \xrightarrow{\alpha[k]} X' \mid Y}$$

(K-PAR-R)

$$\frac{Y \xrightarrow{\alpha[k]} Y' \qquad \mathtt{fresh}(k, X)}{X \mid Y \xrightarrow{\alpha[k]} X \mid Y'}$$

(K-SYN)

$$\frac{X \xrightarrow{\alpha[k]} X' \qquad Y \xrightarrow{\bar{\alpha}[k]} Y' \qquad \alpha \neq \tau}{X \mid Y \xrightarrow{\tau[k]} X' \mid Y'}$$

Figure 7: CCSK forward semantics

Remark 2 The general rule format introduced in [70] does not exploit any structural congruence, including α-conversion. Nevertheless, addition of α-conversion is straightforward by following the instructions of how to deal with structural congruence, given in [70]. To have direct match with RCCS we added α-conversion in our version of CCSK.

Rules for forward transitions are given in Figure 7. Differently from the classic CCS rule for prefix, in the rule K-ACT1, prefix α is not discarded. The fresh new key k is generated and associated with the action α, which after the execution becomes $\alpha[k]$. The annotated prefix $\alpha[k]$ represents the past of the CCSK process $\alpha[k].P$. Rule K-ACT2 inductively allows a prefixed process $\alpha[k].X$ to execute if X can execute. The choice operator is handled by rule K-SUM. As we can notice there is no loss of information by applying this rule. To be more precise, if one of the alternatives, for instance X_j by performing an action $\alpha[k]$ becomes X'_j, then the whole process performs the same action. The other alternatives, need to be a standard (CCS) processes and they do not change. It is necessary to be standard processes to ensure that non of alternatives were executed in the past. Rule K-PAR-L (K-PAR-R) allows left (right) component in the

(K-ACT1$^\bullet$)

$$\alpha[k].P \overset{\alpha[k]}{\rightsquigarrow} \alpha.P$$

(K-ACT2$^\bullet$)

$$\frac{X \overset{\beta[h]}{\rightsquigarrow} X' \qquad k \neq h}{\alpha[k].X \overset{\beta[h]}{\rightsquigarrow} \alpha[k].X'}$$

(K-RES$^\bullet$)

$$\frac{X \overset{\alpha[k]}{\rightsquigarrow} X' \qquad \alpha \notin \{a, \bar{a}\}}{X \backslash a \overset{\alpha[k]}{\rightsquigarrow} X' \backslash a}$$

(K-SUM$^\bullet$)

$$\frac{X_j \overset{\alpha[k]}{\rightsquigarrow} X'_j \ \forall i \neq j. X_i = X'_i \wedge \mathtt{std}(X_i)}{\sum_{i \in I} X_i \overset{\alpha[k]}{\rightsquigarrow} \sum_{i \in I} X'_i}$$

(K-PAR-L$^\bullet$)

$$\frac{X \overset{\alpha[k]}{\rightsquigarrow} X' \qquad \mathtt{fresh}(k, Y)}{X \mid Y \overset{\alpha[k]}{\rightsquigarrow} X' \mid Y}$$

(K-PAR-R$^\bullet$)

$$\frac{Y \overset{\alpha[k]}{\rightsquigarrow} Y' \qquad \mathtt{fresh}(k, X)}{X \mid Y \overset{\alpha[k]}{\rightsquigarrow} X \mid Y'}$$

(K-SYN$^\bullet$)

$$\frac{X \overset{\alpha[k]}{\rightsquigarrow} X' \qquad Y \overset{\bar{\alpha}[k]}{\rightsquigarrow} Y' \qquad \alpha \neq \tau}{X \mid Y \overset{\tau[k]}{\rightsquigarrow} X' \mid Y'}$$

Figure 8: CCSK backward semantics

parallel composition to execute an action $\alpha[k]$ with condition that the key k is not used by the other processes in parallel. Two processes in parallel can communicate by matching the name of the action and key, through the rule K-SYN. Rule K-RES deals with restriction in the classical CCS way. Backward rules are symmetric to the forward ones and they are presented in Figure 8.

To give an intuition how semantics works, let us consider the following example.

Example 5 Let us consider CCS process $P = a.b + c \mid d$ and execution of the action a identified with key k:

$$a.b + c \mid d \xrightarrow{a[k]} a[k].b + c \mid d = X'$$

Note that consumed action a, annotated with the key k, remains in the resulting process. Since choice operator is static, alternative process c is not discarded. The process $a[k].b + c \mid d$ can perform the same forward steps as process $b \mid d$. The backward action removes key k from the process X' and initial process P is obtained.

As in RCCS, we shall consider only reachable processes.

Definition 12 (Reachable Process) *A CCSK process X is* initial *if it is standard, hence a CCS process. A CCSK process X is* reachable *if it can be derived from an initial process P by using the rules in Figures 7 and 8.*

We define the structural property of reachable CCSK processes stating that at most one process among the all alternative processes in the choice operator can be executed, the rest of them must be CCS process. This property is not presented in [23; 43].

Property 1 (Sum form) *If X is a reachable CCSK process and $X = \sum_{i \in I} \pi_i.X_i$, then there exists maximum one index $j \in I$ such that $\pi_j = \alpha_j[k]$. Furthermore, for each $i \neq j$, X_i is a CCS process ($\mathtt{std}(X_i)$ holds).*

Proof The proof is by induction on the length of the derivation from an initial process to the process X, and by case analysis on the last applied rule. □

As for RCCS, the Loop Lemma [70, Proposition 5.1] in CCSK holds.

Lemma 2 (CCSK Loop Lemma) *For each pair of reachable CCSK processes X and X', $X \xrightarrow{\alpha[k]} X'$ iff $X' \overset{\alpha[k]}{\rightsquigarrow} X$.*

The proof follows from the symmetry between forward and backward rules.

Chapter 3

Encodings and Isomorphism between RCCS and CCSK

In the Sections 2.1 and 2.2 we revised two approaches to reversibility in CCS, namely RCCS and CCSK, respectively. Having a two reversible calculi, some natural questions arise: How we can relate these two calculi? Are they equivalent? To answer on this questions we started with the idea of proving bisimilarity between RCCS and CCSK. In [58] we proved that RCCS is at least expressive as CCSK (the proof relies on the encoding of CCSK into RCCS) and presented the idea of the encoding of RCCS into CCSK. In the further work, we redefine both encoding functions presented in [58] with the aim to prove stronger result. In [50] we proved that label transition systems of CCSK and RCCS are isomorphic.

In this Chapter, we show an isomorphism between label transition systems of CCSK and RCCS (up to some structural transformation on processes) relying on two encodings. As an outcome of the isomorphism, these two calculi can be equated by any behavioral equivalence, such as bisilimilarity or trace equivalence. We devise two encodings, one of CCSK into RCCS (Section 3.1) and another one of RCCS into CCSK (Section 3.2) and prove their correctness. The interesting fact about encodings is that

non of them is uniform. Moreover, no uniform encoding can exist since reachable RCCS processes are not closed under parallel composition. Thanks to our results, we show how is possible to bring some properties from one calculus to the other one.

The notion of encoding. Encoding is used as criterion for comparing the expressive power of two languages. There are a different interpretations of what are the properties that good encoding needs to satisfy. In this work we will use the notion of the *uniform* encoding given in [62], stated as follows:

Definition 13 *An encoding* $[\![\cdot]\!]$ *is* uniform *iff it is:*

***homomorphic with respect to the paralel operator*:** *given two processes, P and Q, we have that $(\!|P \mid Q|\!) = (\!|P|\!) \mid (\!|Q|\!)$;*

***renaming preserving*:** *given a process P and an injective renaming function σ which maps names into names, we have that $(\!|P\sigma|\!) = (\!|P|\!)\sigma$.*

The homomorphism of the paralel operator ensures that two paralel processes are translated into two paralel processes, while renaming guarantees that translation does not depend on the channel names.

To prove the correctness of an encoding we shall require that two calculi have the same computations [31]. This idea is formalized through the *operational correspondence* result, which ensures that every computation in one calculi can be mimicked by its translation and that every computation of the translated process corresponds to some computation of the original initial calculi. In the following we give the definition of the operational correspondence.

Process calculus can be represented as a triple $\mathcal{L} = (\mathcal{P}, \rightarrow, \asymp)$, where $\mathcal{P}$ is a set of processes, $\rightarrow$ is an operational semantics and $\asymp$ is behavioural equivalence (usually congruence).

Definition 14 (Operational correspondence) *Given two calculi $\mathcal{L}_i = (\mathcal{P}_i, \rightarrow_i, \asymp_i)$, where $i \in \{1, 2\}$, the encoding $[\![\cdot]\!] : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ is operationally corresponding if holds:*

- *for every transition $P \rightarrow_1 P'$ exist a transition $[\![P]\!] \rightarrow_2 \asymp_2 [\![P']\!]$;*

- *for every transition* $[\![P]\!] \rightarrow_2 Q$*, there exists a process* P' *such that* $P \rightarrow_1 P'$ *and* $Q \rightarrow_2 \asymp_2 [\![P']\!]$

This Chapter is based on the works done in [58] and [50].

Outline. In the Section 3.1 we present encoding of CCSK into RCCS and we prove the operational correspondence. The encoding in the opposite direction (translation of RCCS into CCSK) and its correctness, are shown in Section 3.2. Proof of isomorphism and no existence of uniform encoding are given in Section 3.3. By exploiting a developed theory, we show some cross-fertilisation results in Section 3.4.

3.1 Encoding CCSK into RCCS

In this Section we shall present the encoding of CCSK into RCCS and prove an operational correspondence result.

The main difference between CCSK and RCCS is on how they keep track of the past actions. In RCCS the information about past actions is stored into a memory of a process, while in CCSK the history information is saved in the structure along the whole process. Duplicating the memory every time when having parallel composition in RCCS, has as a consequence that one process in CCSK can correspond to the composition of several monitored processes in RCCS. For instance, CCSK process $X = a[k].(b[w].P \mid Q)$ corresponds to RCCS process

$$R = \langle w, b, \mathbf{0} \rangle \cdot \langle \uparrow \rangle \cdot \langle k, a, \mathbf{0} \rangle \cdot \langle \rangle \triangleright P \mid \langle \uparrow \rangle \cdot \langle k, a, \mathbf{0} \rangle \cdot \langle \rangle \triangleright Q$$

Processes X and R are both derived form the CCS process $a.(b.P \mid Q)$ by executing actions a and b identified with the keys k and w, respectively. As we can notice, process X corresponds to the composition of two parallel monitored processes in R.

Therefore, the encoding needs inductively to examine the structure of the CCSK process X, to able to reconstruct the final memory of the each monitored process in R. Every annotated prefix in X needs to have corresponding memory event in R.

$$
\begin{array}{rcl}
[\![X]\!] & = & [\![X, \langle\rangle]\!] \\
[\![P, m]\!] & = & m \triangleright P \\
[\![\alpha[k].X + \sum_{j\in J} \alpha_j.P_j, m]\!] & = & [\![X, \langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot m]\!] \\
[\![X \mid Y, m]\!] & = & [\![X, \langle\uparrow\rangle \cdot m]\!] \mid [\![Y, \langle\uparrow\rangle \cdot m]\!] \\
[\![X\backslash a, m]\!] & = & [\![X\{^b/_a\}, m]\!]\backslash b \\
 & & \text{if } b \notin \mathtt{fn}(m) \wedge (b = a \vee b \notin \mathtt{fn}(X))
\end{array}
$$

Figure 9: Encoding of CCSK into RCCS

Before giving the definition of encoding function, we would like to recall that P denotes a standard (CCS) process, while X and R stands for CCSK and RCCS process, respectively. We let $\mathcal{P}_K$ and $\mathcal{P}_R$ to be sets of reachable RCCS and CCSK processes.

The encoding function $[\![\cdot]\!] : \mathcal{P}_K \to \mathcal{P}_R$ is defined in terms of an auxiliary encoding function $[\![\cdot]\!] : \mathcal{P}_K \times \mathtt{Mem} \to \mathcal{P}_R$, which takes a RCCS memory as additional argument. For the sake of simplicity we use the same notation for both functions. This does not create any confusion since two function can be easily distinguished by the number of the arguments.

The encoding functions are defined in Figure 9. As we already pointed out, the encoding needs inductively to go through the structure of the CCSK process to be able to built the corresponding memories of the RCCS process. This gives us intuition why is necessary for encoding to take memory m as an additional argument.

The translation of the process X results into calling $[\![X, \langle\rangle]\!]$ with the empty memory as parameter. A standard (CCS) process P with a memory m is encoded as the monitored process $m \triangleright P$, since process P has no computational history.

The encoding of the non standard process X is done by structural induction. Considering the choice operator and thanks to the Property 1, we have that exactly one alternative corresponds to a non standard process.

In our case it will be $\alpha[k].X$, while the rest of the alternatives $\sum_{j\in J} \alpha_j.P_j$ are standard CCS processes. The action event $\langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle$ added to the memory argument consists of the executed action α identified by the key k while the alternative was $\sum_{j\in J} \alpha_j.P_j$.

The parallel and the restriction operators of CCSK are mapped to the corresponding operators of RCCS. In the case of the parallel composition, it is necessary to split encoding of the parallel composition into two independent encodings. At the same time, memory m is duplicated and annotated with $\langle\uparrow\rangle$. In RCCS, through the structural congruence rule RES, restrictions are always pushed on the top of the monitored process. It can be done only under the condition that restricted name is not occurring in the memory of the process (Section 2.1, Figure 5). To capture this behaviour, encoding rule for restriction replaces the name a with the new name b, which is granted to not occur free by the side conditions. In this way the encoding avoid capturing free names. If a is not occur in the memory m, then one can choose $b = a$.

In order to have better intuition of how the encoding works let us consider the following example.

Example 6 Let $X = a + b + c[k].(d[h] \mid P)$. The encoding of X can be computed as:

$$\begin{aligned}
[\![X]\!] &= [\![X, \langle\rangle]\!] \\
&= [\![a + b + c[k].(d[h] \mid P), \langle\rangle]\!] \\
&= [\![d[h] \mid P, \langle k, c, a + b\rangle \cdot \langle\rangle]\!] \\
&= [\![d[h], \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle]\!] \mid [\![P, \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle]\!] \\
&= [\![\mathbf{0}, \langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle]\!] \mid \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle \triangleright P \\
&= \langle h, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle \triangleright \mathbf{0} \mid \langle\uparrow\rangle \cdot \langle k, c, a + b\rangle \cdot \langle\rangle \triangleright P = R
\end{aligned}$$

As we can notice CCSK process X corresponds to the composition of the monitored processes (R).

Remark 3 The renaming of a name in the rule for restriction, makes the encoding nondeterministic in the choice of bound names. This is not an issue since both the calculi feature α-conversion, hence from now on we will consider the encoding as deterministic.

To simplify the technicalities, we make the following assumption stating that sets of the free and bound names of the process $[\![X]\!]$ are disjunctive.

Assumption 1 *We always choose $[\![X]\!]$ in such a way that* $\mathtt{bn}([\![X]\!]) \cap \mathtt{fn}([\![X]\!]) = \emptyset$.

To highlight the necessity for renaming in the rule for restriction, we give the following example:

Example 7 Let $X = a[k].(P \backslash a)$. The encoding of X can be computed as:

$$\begin{aligned}
[\![X]\!] &= [\![X, \langle\rangle]\!] \\
&= [\![a[k].(P \backslash a), \langle\rangle]\!] \\
&= [\![P \backslash a, \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!] \\
&= [\![P\{^b/_a\}, \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!] \backslash b \\
&= (\langle k, a, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P\{^b/_a\}) \backslash b
\end{aligned}$$

We can notice that Assumption 1 is satisfied, since renaming avoids to capture the occurrence of name a in the memory $\langle k, a, \mathbf{0}\rangle \cdot \langle\rangle$.

Before stating the operational correspondence of the encoding, we need some auxiliary results. Any RCCS process R can be seen as a context C^R built from parallel and restriction operators. The context contains numbered holes filled by monitored processes. Therefore, we shall use the following notation:

Notation 2 *We shall write a RCCS process R as $C^R[1 \mapsto m_1 \triangleright P_1, \ldots, n \mapsto m_n \triangleright P_n]$, or more compactly, as $C^R_{i \in \{1,\ldots,n\}}[i \mapsto m_i \triangleright P_i]$. If R is not relevant we may drop it. If $R = C^R[1 \mapsto m_1 \triangleright P_1, \ldots, n \mapsto m_n \triangleright P_n]$ then we write process $R@m$ as $C^R[1 \mapsto m_1@m \triangleright P_1, \ldots, n \mapsto m_n@m \triangleright P_n]$.*

The notation above is used to establish useful properties of the encoding of CCSK processes.

Lemma 3 *Let X be a CCSK process. There exist $C^{[\![X,\langle\rangle]\!]}$, n, m_1, $\ldots$, m_n, P_1, $\ldots$, P_n such that, for each RCCS memory m, we have*

$$[\![X, m]\!] = C^{[\![X,\langle\rangle]\!]}[1 \mapsto m_1@m \triangleright P_1, \ldots, n \mapsto m_n@m \triangleright P_n]$$

Proof The proof is by induction on the derivation of $[\![X, m]\!]$:

- if $X = P$ then by applying the encoding, we have $[\![X, m]\!] = m \triangleright P = C^{[\![X,\langle\rangle]\!]}[m \triangleright P]$ as desired (with $C^{[\![X,\langle\rangle]\!]}[\bullet] = \bullet$, $n = 1$, $m_1 = \langle\rangle$ and $P_1 = P$);

- if $X = \alpha[k].X' + \sum_{j\in J} \alpha_j.Q_j$ then by applying the encoding, we have $[\![X, m]\!] = [\![X', \langle k, \alpha, \sum_{j\in J} \alpha_j.Q_j\rangle \cdot m]\!]$. To simplify the notation, we shall write e instead of $\langle k, \alpha, \sum_{j\in J} \alpha_j.Q_j\rangle$, hence, we have $[\![X, m]\!] = [\![X', e \cdot m]\!]$.

  By inductive hypothesis we get

  $$[\![X', e \cdot m]\!] = C^{[\![X',\langle\rangle]\!]}[1 \mapsto m_1@e \cdot m \triangleright P_1, \ldots, n \mapsto m_n@e \cdot m \triangleright P_n]$$

  as desired, by selecting $C^{[\![X,\langle\rangle]\!]} = C^{[\![X',\langle\rangle]\!]}$;

- if $X = X' \mid X''$ then by applying the encoding, we have $[\![X, m]\!] = [\![X', \langle\uparrow\rangle \cdot m]\!] \mid [\![X'', \langle\uparrow\rangle \cdot m]\!]$, and by inductive hypotheses:

  $$[\![X', \langle\uparrow\rangle \cdot m]\!] = C'^{[\![X',\langle\rangle]\!]}[1 \mapsto m'_1@\langle\uparrow\rangle \cdot m \triangleright P_1, \ldots, n_1 \mapsto m'_{n_1}@\langle\uparrow\rangle \cdot m \triangleright P_{n_1}]$$
  $$[\![X'', \langle\uparrow\rangle \cdot m]\!] = C''^{[\![X'',\langle\rangle]\!]}[1 \mapsto m''_1@\langle\uparrow\rangle \cdot m \triangleright P'_1, \ldots, n_2 \mapsto m''_{n_2}@\langle\uparrow\rangle \cdot m \triangleright P'_{n_2}]$$

  The thesis follows since

  $$\begin{aligned}[\![X, m]\!] = C^{[\![X,\langle\rangle]\!]}[&1 \mapsto m'_1@\langle\uparrow\rangle \cdot m \triangleright P_1, \ldots, n_1 \mapsto m'_{n_1}@\langle\uparrow\rangle \cdot m \triangleright P_{n_1},\\ &n_1 + 1 \mapsto m''_1@\langle\uparrow\rangle \cdot m \triangleright P'_1, \ldots, n_1 + n_2 \mapsto m''_{n_2}@\langle\uparrow\rangle \cdot m \triangleright P'_{n_2}]\end{aligned}$$

  where $C^{[\![X,\langle\rangle]\!]} = C'^{[\![X',\langle\rangle]\!]} \mid C'''^{[\![X'',\langle\rangle]\!]}$ with $C'''^{[\![X'',\langle\rangle]\!]}$ equal to $C''^{[\![X'',\langle\rangle]\!]}$ but for having hole numbers increased by n_1.

- if $X = X'\backslash a$ then by applying the encoding, we have $[\![X'\backslash a, m]\!] = [\![X'\{^b/_a\}, m]\!]\backslash b$ with $b \notin \mathtt{fn}(m) \wedge (b = a \vee b \notin \mathtt{fn}(X))$.
  By inductive hypothesis we have:

  $$[\![X'\{^b/_a\}, m]\!] = C^{[\![X'\{^b/_a\},\langle\rangle]\!]}[1 \mapsto m_1@m \triangleright P_1, \ldots, n \mapsto m_n@m \triangleright P_n]$$

  Hence:

  $$[\![X'\{^b/_a\}, m]\!]\backslash b = C^{[\![X'\{^b/_a\},\langle\rangle]\!]}[1 \mapsto m_1@m \triangleright P_1, \ldots, n \mapsto m_n@m \triangleright P_n]\backslash b$$

  as desired, by selecting $C^{[\![X,\langle\rangle]\!]} = C^{[\![X'\{^b/_a\},\langle\rangle]\!]}\backslash b$.

The thesis follows by noticing that m is only inserted into monitored processes, and has no impact on the other parts of the term. $\square$

In following, we show a property for RCCS transitions which allows us to isolate the impact of structural congruence inside transitions. In other words, we define a relation $\rightarrowtriangle$ which is not influenced by the structural congruence.

Definition 15 *The relation* $\rightarrowtriangle$ *is the smallest relation induced by the rules in Figure 4, left column, except rule* R-EQUIV.

As we can notice from the definitions of the relations $\rightarrowtriangle$ and $\rightarrow$, we have $\rightarrowtriangle \subset \rightarrow$.

Lemma 4 *If there is a RCCS transition* $R \xrightarrow{k,\alpha} S$ *then there exist* $R' \equiv R$ *and* $S' \equiv S$ *such that* $R' \xrightarrowtriangle{k,\alpha} S'$.

Proof The proof is by induction on the derivation of the transition $R \xrightarrow{k,\alpha} S$, with a case analysis on the last applied rule:

- Rule R-ACT: the thesis holds trivially, since R-ACT is an axiom. We can choose $R' = R$, $S' = S$.
- Rule R-PAR-L: we have that $R = R_1 \mid R_2$ and $S = S_1 \mid R_2$, with premise $R_1 \xrightarrow{k,\alpha} S_1$. By inductive hypothesis there exist $R_1' \equiv R_1$ and $S_1' \equiv S_1$ such that $R_1' \xrightarrowtriangle{k,\alpha} S_1'$. By congruence we have $R \equiv R_1' \mid R_2$ and $S \equiv S_1' \mid R_2$, and, by applying rule R-PAR-L with premise $R_1' \xrightarrowtriangle{k,\alpha} S_1'$, we obtain $R_1' \mid R_2 \xrightarrowtriangle{k,\alpha} S_1' \mid R_2$, as desired.
- Rule R-PAR-R: similar to the case above.
- Rule R-SYN: similar to the case above.
- Rule R-RES: we have that $R = R_1 \backslash a$ and $S = S_1 \backslash a$, with premise $R_1 \xrightarrow{k,\alpha} S_1$ and $\alpha \notin \{a, \overline{a}\}$. By inductive hypothesis there exist $R_1' \equiv R_1$ and $S_1' \equiv S_1$ such that $R_1' \xrightarrowtriangle{k,\alpha} S_1'$. By congruence $R \equiv R_1' \backslash a$ and $S \equiv S_1' \backslash a$, and, by applying rule R-RES with premise $R_1' \xrightarrowtriangle{k,\alpha} S_1'$, we obtain $R_1' \backslash a \xrightarrowtriangle{k,\alpha} S_1' \backslash a$, as desired.

- Rule R-EQUIV: we have as premises $R \equiv R_1$, $R_1 \xrightarrow{k,\alpha}\!\!\triangleright S_1$, and $S_1 \equiv S$. By inductive hypothesis there exist $R_1' \equiv R_1$ and $S_1' \equiv S_1$ such that $R_1' \xrightarrow{k,\alpha}\!\!\triangleright S_1'$. We can conclude by noticing that $R \equiv R_1 \equiv R_1'$ and $S \equiv S_1 \equiv S_1'$, as desired. □

In RCCS, the context does not change by performing a forward action without structural congruence influence. The only possibility to change the context is via structural congruence. This is formally stated in the following lemma showing that by executing transition $\rightarrow\!\!\triangleright$, corresponding memory events are added to the parallel components which performed transition, without changing the context.

Lemma 5 *For each forward RCCS transition derived without using rule* R-EQUIV*:*

$$C^R[1 \mapsto m_1 \triangleright P_1, \ldots, n \mapsto m_n \triangleright P_n] \xrightarrow{k,\alpha}\!\!\triangleright S$$

we have:

$$S = C^R[1 \mapsto m_1''@m_1 \triangleright P_1', \ldots, n \mapsto m_n''@m_n \triangleright P_n']$$

where $\langle\uparrow\rangle \notin m_i''$ *for each* $i \in \{1, \ldots, n\}$.

Proof The proof is by induction on the derivation of the transition $\xrightarrow{k,\alpha}\!\!\triangleright$. As the relation $\rightarrow\!\!\triangleright$ is not induced by the rule R-EQUIV, we can conclude that lemma holds by noticing that the RCCS derivation rules only add memories to their monitored processes do not bring any changes to the structure of the context (only structural congruence may change the context). □

The following lemmas are proving that in RCCS, memory has no impact on forward transitions. For instance, if we take two monitored processes $R = m \triangleright \alpha.P$ and $R_1 = m_1 \triangleright \alpha.P$ which differ only for the memories m and m_1, we can notice that both of them can perform action α regardless to the given memories.

This observation is formally stated in the Lemma 8. We prove this results in three steps, first we consider transitions derived without structural congruence, then we examine structural congruence and in the end we combine the two results.

Lemma 6 *There is a forward transition:*

$$C^R[1 \mapsto m_1 \triangleright P_1, \ldots, n \mapsto m_n \triangleright P_n] \xrightarrow{k,\alpha} \\ C^R[1 \mapsto m'_1@m_1 \triangleright P'_1, \ldots, n \mapsto m'_n@m_n \triangleright P'_n]$$

iff there is a forward transition

$$C^R[1 \mapsto \langle\rangle \triangleright P_1, \ldots, n \mapsto \langle\rangle \triangleright P_n] \xrightarrow{k,\alpha} C^R[1 \mapsto m'_1 \triangleright P'_1, \ldots, n \mapsto m'_n \triangleright P'_n]$$

Proof The proof is by rule inspection. Application of the derivation rules for the transition $\rightarrowtriangle$ is not influenced by the memory that monitors the process. □

We can now proceed to the next step and show that memories do not influence structural congruence. The required condition is that name capturing needs to be avoided.

Lemma 7 *If $R \equiv S$ then, for each memory m such that $(\mathtt{bn}(R) \cup \mathtt{bn}(S)) \cap \mathtt{fn}(m) = \emptyset$, $R@m \equiv S@m$.*

Proof By induction on the derivation of $R \equiv S$. The only interesting case is the base one, corresponding to the application of an axiom. We have a case analysis on the applied axiom.

SPLIT: we have $R = m' \triangleright (P \mid Q)$ and $S = \langle\uparrow\rangle \cdot m' \triangleright P \mid \langle\uparrow\rangle \cdot m' \triangleright Q$. By adding the same memory m to the processes R and S we get $R@m = m'@m \triangleright (P \mid Q)$ and $S@m = \langle\uparrow\rangle \cdot m'@m \triangleright P \mid \langle\uparrow\rangle \cdot m'@m \triangleright Q$.

By applying the SPLIT axiom to the process $R@m$ we get $m'@m \triangleright (P \mid Q) \equiv \langle\uparrow\rangle \cdot m'@m \triangleright P \mid \langle\uparrow\rangle \cdot m'@m \triangleright Q$ as desired.

RES: we have $R = m' \triangleright (P\backslash a)$ and $S = (m' \triangleright P)\backslash a$ with $a \notin \mathtt{fn}(m')$. By adding the same memory m to the processes R and S we get $R@m = m'@m \triangleright (P\backslash a)$ and $S@m = (m'@m \triangleright P)\backslash a$.

By applying the axiom RES to the process $R@m$ we get $m'@m \triangleright (P\backslash a) \equiv (m'@m \triangleright P)\backslash a$ as desired. Note that $a \notin \mathtt{fn}(m')$ from the side condition of the inductive hypothesis and $a \notin \mathtt{fn}(m)$ from the statement of the lemma. With these observations, the side condition of the rule RES holds ($a \notin m'@m$).

α: we have $R \equiv S$ since $R =_\alpha S$. By adding the same memory m to both R and S we still have $R@m =_\alpha S@m$ (note that since $(\mathtt{bn}(R) \cup \mathtt{bn}(S)) \cap \mathtt{fn}(m) = \emptyset$ then m is not changed by α-conversion), which implies $R@m \equiv S@m$, as desired. $\square$

Now, we can combine results from the Lemmas 6 and 7, and in following lemma, show that memory does not affect forward transitions.

Lemma 8 *For every forward transition $R \xrightarrow{k,\alpha} R'$ in RCCS and each memory m such that $(\mathtt{bn}(R) \cup \mathtt{bn}(R')) \cap \mathtt{fn}(m) = \emptyset$, we have $R@m \xrightarrow{k,\alpha} R'@m$.*

Proof By applying Lemma 4 we have that there exist S and S' such that $R \equiv S \xrightarrow{k,\alpha} S' \equiv R'$. By α-conversion we can choose S and S' such that $(\mathtt{bn}(S) \cup \mathtt{bn}(S')) \cap \mathtt{fn}(m) = \emptyset$.

Since memories have no impact on structural congruence (Lemma 7) from $R \equiv S$ we can derive $R@m \equiv S@m$. Thanks to Lemma 5, $S \xrightarrow{k,\alpha} S'$ has the form needed by Lemma 6, which by applying, we obtain $S@m \xrightarrow{k,\alpha} S'@m$. Applying again Lemma 7 on $S' \equiv R'$, we can derive $S'@m \equiv R'@m$. By applying rule R-EQUIV we obtain $R@m \xrightarrow{k,\alpha} R'@m$, as desired. $\square$

Now we have all necessary auxiliary lemmas, both forward and backward, to show operational correspondence. With the next two theorems we prove that by having reachable CCSK process X and its encoding $R = [\![X, \langle\rangle]\!]$, if X does an action α in CCSK, then process R does the same action in RCCS. Resulting process in RCCS is equal to the encoding of the resulting process in CCSK. We start by proving forward correctness.

Theorem 1 (Forward Correctness) *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each CCSK transition $X \xrightarrow{\alpha[k]} X'$ there exists a corresponding RCCS transition $R \xrightarrow{k,\alpha} R'$ with $[\![X', \langle\rangle]\!] = R'$.*

Proof The proof is by induction on the derivation of the transition $X \xrightarrow{\alpha[k]} X'$ and by case analysis on the last applied rule.

K-ACT1: We have $\alpha.P \xrightarrow{\alpha[k]} \alpha[k].P$ with $\alpha \in \{a, \bar{a}, \tau\}$. By applying the encoding

$$[\![\alpha.P, \langle\rangle]\!] = \langle\rangle \triangleright \alpha.P$$

Then, by using RCCS rule R-ACT we get $\langle\rangle \triangleright \alpha.P \xrightarrow{k,\alpha} \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P$, with $[\![\alpha[k].P, \langle\rangle]\!] = \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P$.

K-ACT2: We have $\alpha[k].X \xrightarrow{\beta[h]} \alpha[k].X'$ with premise $X \xrightarrow{\beta[h]} X'$. Let $R = [\![X, \langle\rangle]\!]$. By applying the inductive hypothesis we have that $R \xrightarrow{h,\beta} R'$, with $R' = [\![X', \langle\rangle]\!]$. By applying the encoding, we have $[\![\alpha[k].X, \langle\rangle]\!] = [\![X, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!]$ and thanks to Lemma 3 we get $[\![\alpha[k].X, \langle\rangle]\!] = R@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$. Let a be the name in α. By Assumption 1, $a \notin \mathtt{bn}(R) \cup \mathtt{bn}(R')$. By applying Lemma 8 we have that $R@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle \xrightarrow{h,\beta} R'@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$. The thesis follows since $[\![\alpha[k].X, \langle\rangle]\!] = R@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$ and, thanks to Lemma 3, $[\![\alpha[k].X', \langle\rangle]\!] = R'@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$.

K-SUM: We have $\sum_{i\in I} X_i \xrightarrow{\alpha[k]} \sum_{i\in I} X'_i$ with premise $X_j \xrightarrow{\alpha[k]} X'_j$.

We have two cases depending on whether X_j is a CCS process or not. If it is a CCS process then the proof is analogue to the case K-ACT1. If not, it is analogue to the case K-ACT2, with the only difference that the additional memory element has the choice of discarded processes in the last field instead of $\mathbf{0}$.

K-PAR-L: We have $X \mid Y \xrightarrow{\alpha[k]} X' \mid Y$ with premise $X \xrightarrow{\alpha[k]} X'$. Thanks to the definition of the encoding we have that $[\![X \mid Y, \langle\rangle]\!] = [\![X, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![Y, \langle\uparrow\rangle \cdot \langle\rangle]\!]$. Thanks to Lemma 3 we have that

$$[\![X, \langle\uparrow\rangle \cdot \langle\rangle]\!] = [\![X, \langle\rangle]\!]@\langle\uparrow\rangle \cdot \langle\rangle$$

By inductive hypothesis we have that $[\![X, \langle\rangle]\!] = R \xrightarrow{k,\alpha} R' = [\![X', \langle\rangle]\!]$. Thanks to Lemma 8 (the condition is trivially satisfied since $\mathtt{fn}(\langle\uparrow\rangle \cdot \langle\rangle) = \emptyset$) we have that $R@\langle\uparrow\rangle \cdot \langle\rangle \xrightarrow{k,\alpha} R'@\langle\uparrow\rangle \cdot \langle\rangle$. Thanks to Lemma 3 we have $R'@\langle\uparrow\rangle \cdot \langle\rangle = [\![X', \langle\uparrow\rangle \cdot \langle\rangle]\!]$. By applying RCCS rule R-PAR-L we have that

$$[\![X, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![Y, \langle\uparrow\rangle \cdot \langle\rangle]\!] \xrightarrow{k,\alpha} R'@\langle\uparrow\rangle \cdot \langle\rangle \mid [\![Y, \langle\uparrow\rangle \cdot \langle\rangle]\!]$$

We can conclude by noticing that

$$R'@\langle\uparrow\rangle\cdot\langle\rangle \mid [\![Y,\langle\uparrow\rangle\cdot\langle\rangle]\!] = [\![X',\langle\uparrow\rangle\cdot\langle\rangle]\!] \mid [\![Y,\langle\uparrow\rangle\cdot\langle\rangle]\!] = [\![X' \mid Y,\langle\rangle]\!]$$

K-PAR-R: This case is symmetric to the previous one.

K-SYN: We have $X \mid Y \xrightarrow{\tau[k]} X' \mid Y'$ with premises $X \xrightarrow{\alpha[k]} X'$ and $Y \xrightarrow{\bar{\alpha}[k]} Y'$.

Thanks to the definition of the encoding we have that $[\![X \mid Y,\langle\rangle]\!] = [\![X,\langle\uparrow\rangle\cdot\langle\rangle]\!] \mid [\![Y,\langle\uparrow\rangle\cdot\langle\rangle]\!]$. By inductive hypothesis we have that $[\![X,\langle\rangle]\!] \xrightarrow{k,\alpha} R' = [\![X',\langle\rangle]\!]$ and $[\![Y,\langle\rangle]\!] \xrightarrow{k,\overline{\alpha}} S' = [\![Y',\langle\rangle]\!]$. Thanks to Lemma 3 and Lemma 8 (the condition is trivially satisfied since $\mathtt{fn}(\langle\uparrow\rangle\cdot\langle\rangle) = \emptyset$) we have that:

$$[\![X,\langle\uparrow\rangle\cdot\langle\rangle]\!] \xrightarrow{k,\alpha} R'@\langle\uparrow\rangle\cdot\langle\rangle = [\![X',\langle\uparrow\rangle\cdot\langle\rangle]\!]$$

Similarly, we have that:

$$[\![Y,\langle\uparrow\rangle\cdot\langle\rangle]\!] \xrightarrow{k,\overline{\alpha}} S'@\langle\uparrow\rangle\cdot\langle\rangle = [\![Y',\langle\uparrow\rangle\cdot\langle\rangle]\!]$$

By applying RCCS rule R-SYN we have that

$$[\![X,\langle\uparrow\rangle\cdot\langle\rangle]\!] \mid [\![Y,\langle\uparrow\rangle\cdot\langle\rangle]\!] \xrightarrow{k,\tau} [\![X',\langle\uparrow\rangle\cdot\langle\rangle]\!] \mid [\![Y',\langle\uparrow\rangle\cdot\langle\rangle]\!]$$

We can conclude by noticing that

$$[\![X',\langle\uparrow\rangle\cdot\langle\rangle]\!] \mid [\![Y',\langle\uparrow\rangle\cdot\langle\rangle]\!] = [\![X' \mid Y',\langle\rangle]\!]$$

K-RES: We have $X\backslash a \xrightarrow{\alpha[k]} X'\backslash a$ with premise $X \xrightarrow{\alpha[k]} X'$. From the definition of the encoding we have that $[\![X\backslash a,\langle\rangle]\!] = [\![X\{^b/_a\},\langle\rangle]\!]\backslash b$ with $b = a \vee b \notin \mathtt{fn}(X)$. From inductive hypothesis we have that $[\![X,\langle\rangle]\!] \xrightarrow{k,\alpha} [\![X',\langle\rangle]\!]$. Since $\alpha \notin \{a,\bar{a}\}$ we have $[\![X\{^b/_a\},\langle\rangle]\!] \xrightarrow{k,\alpha} [\![X'\{^b/_a\},\langle\rangle]\!]$. By applying RCCS rule R-RES we get

$$[\![X\{^b/_a\},\langle\rangle]\!]\backslash b \xrightarrow{k,\alpha} [\![X'\{^b/_a\},\langle\rangle]\!]\backslash b = [\![X'\backslash a,\langle\rangle]\!]$$

as desired.

α-conversion: We have $X \xrightarrow{\alpha[k]} X'$ with premise $X'' \xrightarrow{\alpha[k]} X'''$, $X =_\alpha X''$ and $X' =_\alpha X'''$. From inductive hypothesis we have that $[\![X'', \langle\rangle]\!] \xrightarrow{k,\alpha} [\![X''', \langle\rangle]\!]$. By applying RCCS α-conversion we get $[\![X, \langle\rangle]\!] \xrightarrow{k,\alpha} [\![X', \langle\rangle]\!]$ as desired. $\square$

Theorem 2 (Backward Correctness) *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each CCSK transition $X \overset{\alpha[k]}{\rightsquigarrow} X'$ there exists a corresponding RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} R'$ with $[\![X', \langle\rangle]\!] = R'$.*

Proof From CCSK Loop Lemma (Lemma 2) we have that $X \overset{\alpha[k]}{\rightsquigarrow} X'$ implies $X' \xrightarrow{\alpha[k]} X$. By Theorem 1 we have that there exists a corresponding RCCS transition $R' \xrightarrow{k,\alpha} R$ with $[\![X, \langle\rangle]\!] = R$ and $[\![X', \langle\rangle]\!] = R'$. By applying RCCS Loop Lemma (Lemma 1) we have that $R \overset{k,\alpha}{\rightsquigarrow} R'$, as desired. $\square$

Up to know, we have proved that every transition done by reachable process X in CCSK can be mimic by its encoding $[\![X, \langle\rangle]\!]$ in RCCS, where resulting process is encoding of the result process in CCSK. Now we need to show the opposite direction, that every transition of the process $R = [\![X, \langle\rangle]\!]$ in RCCS, can be mimic by process X in CCSK, where resulting process in RCCS is congruent with the encoding of the result process in CCSK.

Theorem 3 (Forward Completeness) *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each RCCS transition $R \xrightarrow{k,\alpha} R'$ there exists a corresponding CCSK transition $X \xrightarrow{\alpha[k]} X'$ with $R' \equiv [\![X', \langle\rangle]\!]$.*

Proof Thanks to Lemma 4 we can equivalently write the statement as follows: for each reachable CCSK process X and RCCS processes R and R'', such that $R = [\![X, \langle\rangle]\!]$ and $R'' \equiv R$, if there exists a RCCS transition $R'' \xrightarrow{k,\alpha} R'$, then there exists a corresponding CCSK transition $X \xrightarrow{\alpha[k]} X'$, with $R' \equiv [\![X', \langle\rangle]\!]$. When considering $R'' \equiv R$ we will not consider α-conversion, since it can be trivially matched by CCSK α-conversion.

Now the proof is by structural induction on X with a case analysis on the last applied rule in the derivation of $R'' \xrightarrow{k,\alpha} R'$. We have two main cases, depending on whether X is a standard process P or not.

In the first case $X = P$, and we proceed by performing a structural induction on P.

$P = \alpha.P_1$ **:** we have that $R = [\![\alpha.P_1, \langle\rangle]\!]$ and by applying the encoding

$$[\![\alpha.P_1, \langle\rangle]\!] = \langle\rangle \triangleright \alpha.P_1$$

Since we cannot apply any structural rule (beyond α-conversion), then $R'' = R$.

Then, the only applicable rule is R-ACT, and we get

$$\langle\rangle \triangleright \alpha.P_1 \xrightarrow{k,\alpha} \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P_1 = R'$$

In CCSK, process $\alpha.P_1$ can do the same action α by applying the rule K-ACT1 and we get

$$\alpha.P_1 \xrightarrow{\alpha[k]} \alpha[k].P_1$$

Since $[\![\alpha[k].P_1, \langle\rangle]\!] = R'$ we are done.

$P = \sum_{l\in I} \alpha_l.P_l$ **:** we have $R = [\![\sum_{l\in I} \alpha_l.P_l, \langle\rangle]\!]$. By applying the encoding we have:

$$[\![\sum_{l\in I} \alpha_l.P_l, \langle\rangle]\!] = \langle\rangle \triangleright \sum_{l\in I} \alpha_l.P_l$$

Since no structural congruence (beyond α-conversion) can be applied $R'' = R$. The only applicable rule is R-ACT and we have

$$\langle\rangle \triangleright \sum_{l\in I} \alpha_l.P_l \xrightarrow{k,\alpha_j} \langle k, \alpha_j, \sum_{l\in I\setminus\{j\}} \alpha_l.P_l\rangle \cdot \langle\rangle \triangleright P_j$$

By using CCSK rule K-ACT1 (since P is a CCS process) followed by K-SUM we can derive

$$\sum_{l\in I}\alpha_l.P_l \xrightarrow{\alpha_j[k]} \alpha_j[k].P_j + \sum_{l\in I\setminus\{j\}}\alpha_l.P_l$$

We can conclude by noticing that

$$[\![\alpha_j[k].P_j + \sum_{l\in I\setminus\{j\}}\alpha_l.P_l, \langle\rangle]\!] = [\![P_j, \langle k, \alpha_j, \sum_{l\in I\setminus\{j\}}\alpha_l.P_l\rangle \cdot \langle\rangle]\!] = \langle k, \alpha_j, \sum_{l\in I\setminus\{j\}}\alpha_l.P_l\rangle \cdot \langle\rangle \triangleright P_j$$

as desired.

$P = P_1 \mid P_2$ **:** we have that $R = [\![P_1 \mid P_2, \langle\rangle]\!]$ and by applying the encoding

$$[\![P_1 \mid P_2, \langle\rangle]\!] = \langle\rangle \triangleright P_1 \mid P_2$$

We have now two possibilities for $R'' \equiv R$: either $R'' = R$ or $R'' = \langle\uparrow\rangle \cdot \langle\rangle \triangleright P_1 \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright P_2$. In the first case no rule can be applied and we are done. Let us consider the second one. Now we distinguish three cases depending on whether the last applied rule is R-PAR-L, R-PAR-R or R-SYN.

If R-PAR-L is applied, then we have that

$$\langle\uparrow\rangle \cdot \langle\rangle \triangleright P_1 \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright P_2 \xrightarrow{k,\alpha} R_1 \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright P_2$$

with premise $\langle\uparrow\rangle \cdot \langle\rangle \triangleright P_1 \xrightarrow{k,\alpha} R_1$. Thanks to Lemma 6 we can derive $\langle\rangle \triangleright P_1 \xrightarrow{k,\alpha} R'$ with $R_1 = R'@\langle\uparrow\rangle \cdot \langle\rangle$.

Since $\langle\rangle \triangleright P_1 = [\![P_1, \langle\rangle]\!]$ we can apply the inductive hypothesis and get $P_1 \xrightarrow{\alpha[k]} X'$ with $[\![X', \langle\rangle]\!] = R'$. We can now apply CCSK rule K-PAR-L and derive

$$P_1 \mid P_2 \xrightarrow{\alpha[k]} X' \mid P_2$$

By applying the encoding, we have

$$[\![X' \mid P_2, \langle\rangle]\!] = [\![X', \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![P_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]$$

By applying Lemma 3 on $[\![X', \langle\rangle]\!] = R'$ we can derive $[\![X', \langle\uparrow\rangle \cdot \langle\rangle]\!] = R_1$. We can now conclude that

$$[\![X', \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![P_2, \langle\uparrow\rangle \cdot \langle\rangle]\!] = R_1 \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright P_2$$

as desired.

If R-PAR-R is applied we can reason as in the previous case. If R-SYN is applied, we use twice the inductive hypothesis and we reason as in the previous cases.

$P = P_1 \backslash a$ **:** we have that $R = [\![P_1 \backslash a, \langle\rangle]\!]$ and by applying the encoding

$$[\![P_1 \backslash a, \langle\rangle]\!] = \langle\rangle \triangleright (P_1 \backslash a)$$

We have two possibilities for $R'' \equiv R$: either $R'' = R$ or $R'' = (\langle\rangle \triangleright P_1)\backslash a$. In the first case no rule can be applied and we are done. Let us consider the second one. The only applicable rule is R-RES. We get

$$(\langle\rangle \triangleright P_1)\backslash a \xrightarrow{k,\alpha} R_1 \backslash a$$

with premise $\langle\rangle \triangleright P_1 \xrightarrow{k,\alpha} R_1$, where $\alpha \notin \{a, \overline{a}\}$.

Since $\langle\rangle \triangleright P_1 = [\![P_1, \langle\rangle]\!]$ we can apply the inductive hypothesis and get that $P_1 \xrightarrow{\alpha[k]} X'$ with $[\![X', \langle\rangle]\!] = R_1$. The thesis follows by applying the CCSK rule K-RES on the process $P_1 \backslash a$.

In the second case process X is not standard. We proceed by structural induction on X.

$X = \alpha[k].Y$ **:** we have that $R = [\![\alpha[k].Y, \langle\rangle]\!]$, and by applying the encoding

$$[\![\alpha[k].Y, \langle\rangle]\!] = [\![Y, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!]$$

Take any RCCS transition

$$[\![Y, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!] \equiv R'' \xrightarrow{h,\beta} R'$$

Let a be the name in α. By Assumption 1 $a \notin \mathtt{bn}([\![Y, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!])$. Also, working up to α-conversion we can assume that $a \notin \mathtt{bn}(R'') \cup \mathtt{bn}(R')$.

Since memory has no impact on the structural congruence (Lemma 7) there exists $S'' \equiv [\![Y, \langle\rangle]\!]$ such that $R'' = S''@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$. By Lemma 6 we have that $S'' \xrightarrow{h,\beta} S'$ with $R' = S'@\langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle$. By applying the inductive hypothesis on $[\![Y, \langle\rangle]\!] \equiv S'' \xrightarrow{h,\beta} S'$ we have that $Y \xrightarrow{\beta[h]} Y'$ with $[\![Y', \langle\rangle]\!] \equiv S'$.

By applying CCSK rule K-ACT2 with the premiss $Y \xrightarrow{\beta[h]} Y'$, we can also derive $\alpha[k].Y \xrightarrow{\beta[h]} \alpha[k].Y'$. The thesis follows since $[\![\alpha[k].Y', \langle\rangle]\!] = [\![Y', \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!] \equiv R'$ thanks to Lemma 3.

$X = \alpha_j[k].X_j + \sum_{l \in I \setminus \{j\}} X_l$ **:** we have that $R = [\![\alpha_j[k].X_j + \sum_{l \in I \setminus \{j\}} X_l, \langle\rangle]\!]$, and by applying the encoding

$$[\![\alpha_j[k].X_j + \sum_{l \in I \setminus \{j\}} X_l, \langle\rangle]\!] = [\![X_j, \langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle]\!]$$

Take any RCCS transition

$$[\![X_j, \langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle]\!] \equiv R'' \xrightarrow{h,\beta} R'$$

Let $R''' = [\![X_j, \langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle]\!]$. From Assumption 1 we have that $\mathtt{fn}(\langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle) \cap \mathtt{bn}(R''') = \emptyset$. Also, working up to α-conversion we can assume that $\mathtt{fn}(\langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle) \cap (\mathtt{bn}(R'') \cup \mathtt{bn}(R')) = \emptyset$.

Since memory has no impact on structural congruence (Lemma 7) there exists $S'' \equiv [\![X_j, \langle\rangle]\!]$ such that $R'' = S''@\langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle$. By Lemma 6 we have that $S'' \xrightarrow{h,\beta} S'$ with $R' = S'@\langle k, \alpha_j, \sum_{l \in I \setminus \{j\}} X_l\rangle \cdot \langle\rangle$. By applying the inductive hypothesis on $[\![X_j, \langle\rangle]\!] \equiv S'' \xrightarrow{h,\beta} S'$ we have that $X_j \xrightarrow{\beta[h]} X'_j$ with $[\![X'_j, \langle\rangle]\!] \equiv S'$.

By applying CCSK rule K-ACT2 and K-SUM we can also derive

$$\alpha_j[k].X_j + \sum_{l \in L \setminus \{j\}} X_l \xrightarrow{\beta[h]} \alpha_j[k].X'_j + \sum_{l \in L \setminus \{j\}} X_l$$

The thesis follows since

$$[\![\alpha_j[k].X'_j + \sum_{l\in L\setminus\{j\}} X_l, \langle\rangle]\!] = [\![X'_j, \langle k, \alpha_j, \sum_{l\in L\setminus\{j\}} X_l\rangle \cdot \langle\rangle]\!] \equiv R'$$

thanks to Lemma 3.

$X = Y_1 \mid Y_2$ **:** we have that $R = [\![Y_1 \mid Y_2, \langle\rangle]\!]$, and by applying the encoding we obtain

$$[\![Y_1 \mid Y_2, \langle\rangle]\!] = [\![Y_1, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![Y_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]$$

Take any term $R'' \equiv [\![Y_1, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![Y_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]$. There are two cases: either $R'' = R''_1 \mid R''_2$ with $R''_1 \equiv [\![Y_1, \langle\uparrow\rangle \cdot \langle\rangle]\!]$ and $R''_2 \equiv [\![Y_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]$, or the two parallel sub-processes have been merged by applying the (SPLIT) rule from right to left. In this last case no transition can be performed. Let us consider the first case. We have a case analysis depending on whether the last applied rule is R-PAR-L, R-PAR-R or R-SYN.

If rule R-PAR-L is applied we have that $R''_1 \mid R''_2 \xrightarrow{k,\alpha} S''_1 \mid R''_2$ with hypothesis $R''_1 \xrightarrow{k,\alpha} S''_1$. Moreover, from Lemma 6 (condition is verified since $\mathtt{fn}(\langle\uparrow\rangle \cdot \langle\rangle) = \emptyset$) there exist R'''_1 such that $R''_1 = R'''_1@\langle\uparrow\rangle \cdot \langle\rangle$ and $R'''_1 \xrightarrow{k,\alpha} S'''_1$, where $S''_1 = S'''_1@\langle\uparrow\rangle \cdot \langle\rangle$.

Since $R'''_1 \equiv [\![Y_1, \langle\rangle]\!]$ we can apply the inductive hypothesis and obtain that

$$Y_1 \xrightarrow{\alpha[k]} Y'_1 \text{ with } [\![Y'_1, \langle\rangle]\!] \equiv S'''_1$$

We can now apply CCSK rule K-PAR-L and derive the transition

$$Y_1 \mid Y_2 \xrightarrow{\alpha[k]} Y'_1 \mid Y_2$$

and we conclude by noticing that

$$[\![Y'_1 \mid Y_2, \langle\rangle]\!] = [\![Y'_1, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![Y_2, \langle\uparrow\rangle \cdot \langle\rangle]\!] \equiv S''_1 \mid R''_2$$

The other two cases are similar.

$X = Y\backslash a$ **:** we have that $R = [\![Y\backslash a, \langle\rangle]\!]$ and by applying the encoding $[\![Y\backslash a, \langle\rangle]\!] = [\![Y\{^b/_a\}, \langle\rangle]\!]\backslash b$ with $b = a \vee b \notin \mathtt{fn}(Y)$. Take any term $R'' \equiv [\![Y\{^b/_a\}, \langle\rangle]\!]\backslash b$. There are two cases: either $R'' = R''_1\backslash b$ with $R''_1 \equiv [\![Y\{^b/_a\}, \langle\rangle]\!]$, or the restriction has been put back inside the term using structural rule RES. In this last case no transition can be performed. Let us consider the first case. The only applicable rule is R-RES. We have $R''_1\backslash b \xrightarrow{k,\alpha} R'''\backslash b$ with hypothesis $R''_1 \xrightarrow{k,\alpha} R'''$. By applying the inductive hypothesis we obtain that

$$Y\{^b/_a\} \xrightarrow{\alpha[k]} Y'\{^b/_a\}$$

with $Y'\{^b/_a\} \equiv R'''$. By applying CCSK rule K-RES we also have

$$Y\{^b/_a\}\backslash b \xrightarrow{\alpha[k]} Y'\{^b/_a\}\backslash b$$

The thesis follows since $[\![Y'\backslash a, \langle\rangle]\!] = [\![Y'\{^b/_a\}, \langle\rangle]\!]\backslash b \equiv R'''\backslash b$. □

Theorem 4 (Backward Completeness) *Let X be a reachable CCSK process and $R = [\![X, \langle\rangle]\!]$. For each RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} R'$ there exists a corresponding CCSK transition $X \overset{\alpha[k]}{\rightsquigarrow} X'$ with $R' \equiv [\![X', \langle\rangle]\!]$.*

Proof From RCCS Loop Lemma (Lemma 1) we have that $R' \xrightarrow{k,\alpha} R$ in RCCS. From Theorem 3 we have that $X' \xrightarrow{\alpha[k]} X$ with $[\![X, \langle\rangle]\!] \equiv R$ and $[\![X', \langle\rangle]\!] = R'$. By applying RCCS rule R-EQUIV to $R' \xrightarrow{k,\alpha} R$ and $R \equiv [\![X, \langle\rangle]\!]$ we have that $R' \xrightarrow{k,\alpha} [\![X, \langle\rangle]\!]$. By applying CCSK Loop Lemma (Lemma 2) we have that $X \overset{\alpha[k]}{\rightsquigarrow} X'$. □

3.2 Encoding RCCS into CCSK

In this Section we give the first encoding of RCCS into CCSK, appeared in [58] and discuss a difficulties behind it. After that we present the new encoding function and prove operational correspondence result.

As we already pointed out, the main difference between RCCS and CCSK is in the way how they keep track of the past actions. To be able

to manipulate over monitored processes in RCCS it is necessary to use the additional structural congruence rules. One of them is structural rule SPLIT which split a parallel composition of processes sharing the same memory into a parallel composition of different monitored processes. In this way, the obtained processes can evolve independently. In the CCSK, history information is saved in the structure of the process and there is no need for the structural rules. Therefore, the encoding of RCCS has to be able, to collect partially encoded processes which have the same part of the memory and to merge them. For instance, let us take RCCS process

$$R = \langle h, \beta, \mathbf{0} \rangle \cdot \langle \uparrow \rangle \cdot \langle k, \alpha, Q \rangle \cdot \langle \rangle \triangleright P_1 \mid \langle \uparrow \rangle \cdot \langle k, \alpha, Q \rangle \cdot \langle \rangle \triangleright \gamma.P_2$$

As we mentioned, the encoding function, while translating process R needs to be aware that two subprocesses have equal part of the memory ($\langle \uparrow \rangle \cdot \langle k, \alpha, Q \rangle \cdot \langle \rangle$) and to join them. To be able to do that, and to obtain corresponding process $X = \alpha[k].(\beta[h].P_1 \mid \gamma.P_2) + Q$ in CCSK, encoding has to reason not just on the every component of the parallel composition separately, but on the whole process as well.

There are two ways of looking into the memory of a RCCS process: either left (last action) or right (very first action). If we look from the left then the encoding is:

Encoding from the left. Taking into account structure of the monitored process and observation stated above, we define translation of RCCS process through two encoding functions. They encode a memory of the monitored process starting from the very last memory event (left in the textual representation). The function $(\!|\cdot|\!)$ will encode monitored processes on the local level and it is inductively defined as follows:

$$\begin{array}{ll}
(\!|R \mid S|\!) = (\!|R|\!) \mid (\!|S|\!) & (\!|\langle\rangle, X|\!) = X \\
(\!|R \backslash A|\!) = (\!|R|\!) \backslash A & (\!|\langle k, \alpha, Q \rangle \cdot m, X|\!) = (\!|m, \alpha[k].X + Q|\!) \\
(\!|m \triangleright P|\!) = (\!|m, P|\!) & (\!|\langle \uparrow \rangle \cdot m, X|\!) = \wr \langle \uparrow \rangle \cdot m, X \wr
\end{array}$$

As we can see, the encoding of a monitored process $(\!|m, P|\!)$ proceeds as long as in memory m it does not encounter a split event $\langle \uparrow \rangle$. In that

state, encoding freezes and produces partially encoded process of the form $\wr\langle\uparrow\rangle \cdot m, X\int$.

To continue with encoding, it is necessary to track down another partially encoded process $\wr\langle\uparrow\rangle \cdot m, Y\int$ with the same memory and join them into parallel composition with shared memory $\wr m, X \mid Y\int$. For this purpose we introduce the function $\delta(\cdot)$ which will reason on the global level and be in charge of putting together two partially encoded CCSK processes sharing the same memory. The function $\delta(\cdot)$ is defined as follows:

$$\delta\left(\prod \wr\langle\uparrow\rangle \cdot m_l, X_l\int \mid \prod \wr\langle\uparrow\rangle \cdot m_t, X_t\int \mid \prod \wr\langle\uparrow\rangle \cdot m_z, X_z\int\right) \quad =$$
$$\delta\left(\prod (\!| m_l, X_l \mid X_t |\!) \mid \prod \wr\langle\uparrow\rangle \cdot m_z, X_z\int\right) \quad \text{if } \forall l \in L\, \exists t \in T \text{ s.t } m_l = m_t$$
$$\delta(X) = X$$

The function δ applies again the encoding $(\!|\cdot|\!)$ on the joined processes until an entire CCSK process has been obtained.

In the definition of encoding functions stated above, naturally, we were thinking about translation of the memory of the monitored process starting from the left to the right, i.e. starting from the element on the top of the stack. This way of reasoning, even by redefining existing functions, is showed to be unsuitable for proving operational correspondence. For this reason, we took a different approach, presented in the further text.

Encoding of RCCS into CCSK As we stated before, the encoding needs to translate together the monitored processes having the same memory part m. Since in RCCS, memory is duplicated every time when monitored process has parallel composition on the top level, encoding of the memory m should start from the very first action that process did i.e. the oldest action (right in the textual representation), differently from the initial encoding definition given in the paragraph above. To illustrate this, we give the following example.

Example 8 Let us take the RCCS process

$$R = \langle h, \beta, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\rangle \triangleright P_1 \mid \langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\rangle \triangleright P_2$$

Instead to start to encode memories of the monitored subprocesses from the left in the textual representation, one should start from the right by taking the part of the memory that is the same for both subprocesses, that is $\langle k, \alpha, Q\rangle$ and build from it the CCSK context $C = \alpha[k].[\bullet] + Q$. The translation of the rest of the memories continue inside the context C.

Before stating the definition of the encoding, we need to introduce an auxiliary function and a history context. We define a trimming function δ to remove the split events $\langle\uparrow\rangle$ from the memory, starting from the right side of it. For instance, the result of applying δ to the memory $\langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle$ is $\langle\uparrow\rangle \cdot \langle k, \alpha, Q\rangle \cdot \langle\rangle$. The trimming function δ is defined as follows:

Definition 16 *The function $\delta : \mathtt{Mem} \to \mathtt{Mem}$, is inductively defined as follows:*

$$\delta(m@\langle\uparrow\rangle \cdot \langle\rangle) = \delta(m) \qquad\qquad \delta(\langle\rangle) = \langle\rangle$$
$$\delta(m@\langle k, \alpha, Q\rangle \cdot \langle\rangle) = m@\langle k, \alpha, Q\rangle \cdot \langle\rangle$$

Now we introduce the notion of *history context* which represents the correspondence between RCCS memory without $\langle\uparrow\rangle$ elements and the CCSK context.

Definition 17 (History Context) *Given a memory m such that $\langle\uparrow\rangle \notin m$, the corresponding* history context $\mathtt{H}_m$ *is defined as follows:*

$$\begin{array}{rcl} \mathtt{H}_{\langle\rangle} & = & \bullet \\ \mathtt{H}_{\langle k,\alpha,\mathbf{0}\rangle\cdot m} & = & \mathtt{H}_m[\alpha[k].\bullet] \\ \mathtt{H}_{\langle k,\alpha,Q\rangle\cdot m} & = & \mathtt{H}_m[\alpha[k].\bullet + Q] \textit{ if } Q \neq \mathbf{0} \end{array}$$

To illustrate definition above, consider RCCS process $\langle h, b, \mathbf{0}\rangle \cdot \langle k, a, Q\rangle \triangleright P$. History context, of the given memory $\langle h, b, \mathbf{0}\rangle \cdot \langle k, a, Q\rangle$, is defined as

$$\mathtt{H}_{\langle h,b,\mathbf{0}\rangle\cdot\langle k,a,Q\rangle}[P] = \mathtt{H}_{\langle k,a,Q\rangle}[b[h].P] = \mathtt{H}_{\langle\rangle}[a[k].b[h].P + Q]$$

Finally, we can give a definition of the encoding function. Let us recall that P is a standard CCS process and $\mathcal{P}_K$ and $\mathcal{P}_R$ are the sets of, CCSK and RCCS processes, respectively. The encoding function $(\!|\cdot|\!) : \mathcal{P}_R \to \mathcal{P}_K$

is inductively defined as follows:

$$
\begin{aligned}
(\!|R\backslash a|\!) &= (\!|R|\!)\backslash a \\
(\!|C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m) \triangleright P_l]|\!) &= \mathtt{H}_m[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]] \\
&\quad \text{if } \langle\uparrow\rangle \notin m \text{ and the top operator in } C \text{ is } | \\
(\!|m \triangleright P|\!) &= \mathtt{H}_m[P]
\end{aligned}
$$

Let us comment on the rules. The rule for restriction is design to push restriction outside of the encoding. Collection of the common parts of the memories is done by the function $(\!|\cdot|\!)$. It examines the memory starting form the oldest element (right in the textual representation) and takes the common part m such that $\langle\uparrow\rangle \notin m$. The common memory m is used for reconstructing the CCSK structure by history context $\mathtt{H}_m$. The first split element from the right is discarded and the rest of the memories m_l are being trimmed by function δ and kept for further encoding. In this way, processes that were split by using SPLIT rule in RCCS, are merged under the common memory. Single monitored process $m \triangleright P$ is encoded by translating the memory m into the history context $\mathtt{H}_m$, which builds the history part of the CCSK process.

We would like to remark that in the second rule, notation C is used to represent contexts in both calculi, CCSK and RCCS. We justify this abuse of notation with the fact that context C is composed by parallel and restriction operators, which are available in both calculi.

The intuition how encoding works is given in the following example.

Example 9 Let us take the RCCS process R:

$$R = \langle k, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright a \mid \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright b \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright c$$

Its encoding in CCSK is:

$$
\begin{aligned}
(\!|R|\!) &= (\!|\langle k, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright a \mid \langle\uparrow\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle \triangleright b \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright c|\!) && (3.1)\\
&= \mathtt{H}_{\langle\rangle}[(\!|\delta(\langle k, d, \mathbf{0}\rangle \cdot \langle\uparrow\rangle \cdot \langle\rangle) \triangleright a|\!) \mid (\!|\delta(\langle\uparrow\rangle \cdot \langle\rangle) \triangleright b|\!) \mid (\!|\delta(\langle\rangle) \triangleright c|\!)] && (3.2)\\
&= (\!|\langle k, d, \mathbf{0}\rangle \cdot \langle\rangle \triangleright a|\!) \mid (\!|\langle\rangle \triangleright b|\!) \mid (\!|\langle\rangle \triangleright c|\!) && (3.3)\\
&= \mathtt{H}_{\langle k,d,\mathbf{0}\rangle\cdot\langle\rangle}[a] \mid \mathtt{H}_{\langle\rangle}[b] \mid \mathtt{H}_{\langle\rangle}[c] && (3.4)\\
&= \mathtt{H}_{\langle\rangle}[d[k].a] \mid b \mid c && (3.5)\\
&= d[k].a \mid b \mid c && (3.6)
\end{aligned}
$$

In (3.2), we can notice that the memory part that was the same for all three subprocesses is only the empty memory $\langle\rangle$ which build the context $\mathtt{H}_{\langle\rangle}$. The first split event from the right, is consumed by encoding function and removed from each memory. The translation of the rest of the memories continue inside of the context $\mathtt{H}_{\langle\rangle}$.

In what follows we show some auxiliary results, necessary to prove the operational correspondence of the encoding.

Definition 18 *Given a context $C_{l\in L}[l \mapsto \bullet_l]$ we denote with h_l the number of parallel composition operators in the path connecting $\bullet_l$ to the root in the syntax tree of $C_{l\in L}[l \mapsto \bullet_l]$.*

The following lemma states that history context with some memory m can be decomposed into two contexts with the memories m_1 and m_2 where $m = m_1@m_2$.

Lemma 9 *For any memory $m_1@m_2$, such that $\langle\uparrow\rangle \notin m_1, m_2$, and reachable CCSK process X, we have that $\mathtt{H}_{m_1@m_2}[X] = \mathtt{H}_{m_2}[\mathtt{H}_{m_1}[X]]$*

Proof The proof is by induction on the length of m_1.

- The base case is when $m_1 = \langle\rangle$. In this case:

  $$\mathtt{H}_{m_1@m_2}[X] = \mathtt{H}_{m_2}[X]$$

  since $m_1@m_2 = m_2$ and:

  $$\mathtt{H}_{m_2}[\mathtt{H}_{m_1}[X]] = \mathtt{H}_{m_2}[X]$$

  since $\mathtt{H}_{m_1} = \bullet$. The thesis follows.

- The inductive case is when $m_1 = \langle k, \alpha, \mathbf{0} \rangle \cdot m_1'$. We have that:

$$\mathtt{H}_{m_1@m_2}[X] = \mathtt{H}_{(\langle k,\alpha,\mathbf{0}\rangle \cdot m_1')@m_2}[X] = \mathtt{H}_{m_1'@m_2}[\alpha[k].X]$$

  By applying inductive hypothesis on $m_1'@m_2$ we have:

$$\mathtt{H}_{m_1'@m_2}[\alpha[k].X] = \mathtt{H}_{m_2}[\mathtt{H}_{m_1'}[\alpha[k].X]] = \mathtt{H}_{m_2}[\mathtt{H}_{\langle k,\alpha,\mathbf{0}\rangle \cdot m_1'}[X]]$$

  as desired.

  The case where we have Q instead of $\mathbf{0}$ is analogous. □

Lemma 10 *For each reachable RCCS process of the form $C_{l \in L}[l \mapsto m_l' \triangleright P_l]$, if $m_l' = m_l@\langle\uparrow\rangle \cdot m_l''$ where $\langle\uparrow\rangle \notin m_l''$ then we have*

$$m_l = \delta(m_l)@(\langle\uparrow\rangle^{h_l - 1} \cdot \langle\rangle)$$

Proof By induction on the number of steps leading from an initial RCCS process to $C_{l \in L}[l \mapsto m_l' \triangleright P_l]$.

- The base case is trivial, since the memory of an initial process never contains $\langle\uparrow\rangle$ elements (memory of an initial process is empty, $\langle\rangle$). Let us consider the inductive case.

- If the reduction is derived without using structural congruence, the thesis follows by inductive hypothesis using Lemma 5.

  If structural congruence is used, let us consider the application of a single axiom. If the axiom is (RES) or (α), then the thesis follows by inductive hypothesis. If the axiom is (SPLIT), then a memory with depth h_l is split into two memories with depth $h_l + 1$, and a $\langle\uparrow\rangle$ element is added to both of them. For all the other memories the depth is unchanged and no $\langle\uparrow\rangle$ element is added. The thesis follows by inductive hypothesis. □

Lemma 11 *For each reachable CCSK process X and history context $\mathtt{H}_m$ we have*

$$[\![\mathtt{H}_m[X], \langle\rangle]\!] = [\![X, \langle\rangle]\!]@m$$

Proof The proof is by structural induction on m.

- The base case when $m = \langle\rangle$ is trivial.
- Let us consider the case $m = \langle k, \alpha, \mathbf{0}\rangle \cdot m'$ (the case with Q instead of $\mathbf{0}$ is analogous). By definition of history context we have

  $$[\![\mathtt{H}_{\langle k,\alpha,\mathbf{0}\rangle\cdot m'}[X], \langle\rangle]\!] = [\![\mathtt{H}_{m'}[\alpha[k].X], \langle\rangle]\!]$$

  By inductive hypothesis we have

  $$[\![\mathtt{H}_{m'}[\alpha[k].X], \langle\rangle]\!] = [\![\alpha[k].X, \langle\rangle]\!]@m'$$

  From the definition of the $[\![\cdot,\cdot]\!]$ encoding we get

  $$[\![\alpha[k].X, \langle\rangle]\!]@m' = [\![X, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!]@m'$$

  From Lemma 3 we have

  $$[\![X, \langle k, \alpha, \mathbf{0}\rangle \cdot \langle\rangle]\!]@m' = [\![X, \langle\rangle]\!]@(\langle k, \alpha, \mathbf{0}\rangle \cdot m')$$

  as desired. □

Now we can show the main result of this Section stating that RCCS is more abstract than CCSK. The proof of the result is done in two steps. First, we show that by taking an RCCS process R and encoding it to CCSK and encoding result back in RCCS, we obtain the initial process R. After that we prove that if we start from a CCSK process X, encode it to RCCS and then back to CCSK we obtain normalisation of X, not exactly the process X from which we started.

Theorem 5 *Let R be a reachable RCCS process. Then $[\![(\!|R|\!), \langle\rangle]\!] = R$.*

Proof The proof is by induction on the derivation of $(\!|R|\!)$.

$(\!|R\backslash a|\!)$ **:** by using the definitions of the encodings we get

$$[\![(\!|R\backslash a|\!), \langle\rangle]\!] = [\![(\!|R|\!)\backslash a, \langle\rangle]\!] = [\![(\!|R|\!), \langle\rangle]\!]\backslash a$$

since $a \notin \mathtt{fn}(\langle\rangle)$, and by applying the inductive hypothesis we have that $[\![(\!|R|\!), \langle\rangle]\!]\backslash a = R\backslash a$, as desired.

$(\!| C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m) \triangleright P_l] |\!)$ with $\langle\uparrow\rangle \notin m$: by using the definition of the encoding $(\!|\cdot|\!)$

$$[\![(\!| C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m) \triangleright P_l] |\!), \langle\rangle]\!] = [\![\mathtt{H}_m[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]], \langle\rangle]\!]$$

By using Lemma 11 we have:

$$[\![\mathtt{H}_m[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]], \langle\rangle]\!] = [\![C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)], \langle\rangle]\!]@m$$

From the definition of $[\![\cdot,\cdot]\!]$ we have:

$$[\![C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)], \langle\rangle]\!]@m = (C_{l\in L}[l \mapsto [\![(\!|\delta(m_l) \triangleright P_l|\!), \langle\uparrow\rangle^{h_l} \cdot \langle\rangle]\!]])@m$$

By applying Lemma 3 we have:

$$(C_{l\in L}[l \mapsto [\![(\!|\delta(m_l) \triangleright P_l|\!), \langle\uparrow\rangle^{h_l} \cdot \langle\rangle]\!]])@m = \\ (C_{l\in L}[l \mapsto [\![(\!|\delta(m_l) \triangleright P_l|\!), \langle\rangle]\!]@\langle\uparrow\rangle^{h_l} \cdot \langle\rangle])@m$$

By inductive hypothesis we have:

$$(C_{l\in L}[l \mapsto [\![(\!|\delta(m_l) \triangleright P_l|\!), \langle\rangle]\!]@\langle\uparrow\rangle^{h_l} \cdot \langle\rangle])@m = \\ (C_{l\in L}[l \mapsto (\delta(m_l) \triangleright P_l)@\langle\uparrow\rangle^{h_l} \cdot \langle\rangle])@m$$

By definition of append we have:

$$(C_{l\in L}[l \mapsto (\delta(m_l) \triangleright P_l)@\langle\uparrow\rangle^{h_l} \cdot \langle\rangle])@m = C_{l\in L}[l \mapsto \delta(m_l)@(\langle\uparrow\rangle^{h_l} \cdot \langle\rangle)@m \triangleright P_l]$$

The thesis follows thanks to Lemma 10.

$(\!|m \triangleright P|\!)$ **:** by definition of $(\!|\cdot|\!)$ we have $(\!|m \triangleright P|\!) = \mathtt{H}_m[P]$. By applying $[\![\cdot, \langle\rangle]\!]$ we get:

$$[\![(\!|m \triangleright P|\!), \langle\rangle]\!] = [\![\mathtt{H}_m[P], \langle\rangle]\!]$$

By applying Lemma 11 and the definition of the $[\![\cdot,\cdot]\!]$ encoding we have:

$$[\![\mathtt{H}_m[P], \langle\rangle]\!] = [\![P, \langle\rangle]\!]@m = (\langle\rangle \triangleright P)@m = m \triangleright P$$

as desired. □

To prove the converse of Theorem above (Theorem 5), we need to introduce the notion of a *normal form* for CCSK processes. Essentially, normal form pushes all the restrictions in the non-standard part of a sequential process on the top level. Precisely, normal form of the CCSK process is defined as:

Definition 19 (CCSK normal form) *The normal form* $\mathtt{nf}(X)$ *of a CCSK process* X *is a CCSK process* Y *obtained from* X *by applying as many times as possible the following rewriting rules (in any context):*

- $\alpha[k].(X_1\backslash a) \to (\alpha[k].X_1\{^b/_a\})\backslash b$ *if* $b \notin \mathtt{fn}(\alpha.\mathbf{0}) \wedge (b = a \vee b \notin \mathtt{fn}(X_1))$
- $\alpha[k].(X_1\backslash a) + Q \to (\alpha[k].X_1\{^b/_a\} + Q)\backslash b$
  if $b \notin \mathtt{fn}(\alpha.Q) \wedge (b = a \vee b \notin \mathtt{fn}(X_1))$

where X_1 *is not a standard process.*

Let us note that $\mathtt{nf}(\cdot)$ avoid capturing free names, mimicking the behaviour of the encoding $[\![\cdot]\!]$. In the following examples, we show the necessity for the normal form $\mathtt{nf}(\cdot)$.

Example 10 Let us consider the CCSK process $X = a[k].(b[w].P)\backslash a$. Its encoding into CCSK is:

$$\begin{aligned}
[\![a[k].(b[w].P)\backslash a, \langle\rangle]\!] &= [\![(b[w].P)\backslash a, \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!] \\
&= [\![(b[w].P\{^c/_a\}), \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!]\backslash c \\
&= [\![P\{^c/_a\}, \langle w, b, \mathbf{0}\rangle \cdot \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!]\backslash c \\
&= (\langle w, b, \mathbf{0}\rangle \cdot \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle \triangleright P\{^c/_a\})\backslash c = \ R
\end{aligned}$$

Let $m = \langle w, b, \mathbf{0}\rangle \cdot \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle$. By applying the encoding $(\!|\cdot|\!)$ on the process R, we obtain:

$$\begin{aligned}
(\!|(m \triangleright P\{^c/_a\})\backslash c|\!) &= (\!|(m \triangleright P\{^c/_a\})|\!)\backslash c \\
&= \mathtt{H}_m[P\{^c/_a\}]\backslash c \\
&= (a[k].b[w].P\{^c/_a\})\backslash c
\end{aligned}$$

We have that initial CCSK process $X \neq (a[k].b[w].P\{^c/_a\})\backslash c$, but $\mathtt{nf}(X) =_\alpha (a[k].b[w].P\{^c/_a\})\backslash c$.

Remark 4 We would like to highlight the fact that reduction to normal form is the identity if all the restrictions are in the standard part of the process. For example, consider a CCSK process $X = a[k].(P\backslash a)$. Its encoding to RCCS is:

$$[\![a[k].(P\backslash a), \langle\rangle]\!] = [\![P\backslash a, \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle]\!] = \langle k, a, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (P\backslash a)$$

If we apply the encoding $(\!|\cdot|\!)$ to $\langle k, a, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (P\backslash a)$ we obtain:

$$(\!|\langle k, a, \mathbf{0}\rangle \cdot \langle\rangle \triangleright (P\backslash a)|\!) = \mathtt{H}_{\langle k,a,\mathbf{0}\rangle\cdot\langle\rangle}[P\backslash a] = a[k].(P\backslash a) = X = \mathtt{nf}(X)$$

If we use a different definition of normal form $\mathtt{nf}^{\bullet}(\cdot)$that allows to push out restrictions also from the standard part of processes (we could done it by removing the part of the Definition 19 stating that X_1 cannot be a standard process), we would have that:

$$\mathtt{nf}^{\bullet}(X) = (a[k].P\{^{b}/_{a}\})\backslash b$$

but in that case $(\!|[\![X, \langle\rangle]\!]|\!) \neq \mathtt{nf}^{\bullet}(X)$ and this is not what we wanted.

We shall highlight that processes with the same normal form have the same behaviour. This is shown by the following lemma.

Lemma 12 *Let X and Y be CCSK processes such that $\mathtt{nf}(X) = \mathtt{nf}(Y)$. Then $X \xrightarrow{\alpha[k]} X'$ iff $Y \xrightarrow{\alpha[k]} Y'$ with $\mathtt{nf}(X') = \mathtt{nf}(Y')$, and similarly for backward transitions.*

Proof The proof is by induction on the number of the application of the rewriting rules used to compute the normal form, both in the direction used for normalization and in the opposite direction. From the definition of the normal form (Definition 19) it is easy to see that lemma holds for a single application of the rules. Inductive step follows directly form the inductive hypothesis and the fact that lemma is holding for a single application of the rewriting rule. □

With the next lemma we show some properties of the encoding function $(\!|\cdot|\!)$.

Lemma 13 *For each memory m and reachable RCCS processes R and S we have:*

1. $(\!|R@m|\!) = \mathtt{nf}(\mathtt{H}_m[(\!|R|\!)])$ *where* $\langle\uparrow\rangle \notin m$
2. $(\!|R@\langle\uparrow\rangle \cdot \langle\rangle \mid S@\langle\uparrow\rangle \cdot \langle\rangle|\!) = (\!|R|\!) \mid (\!|S|\!)$

Proof The proof of item (1) is by structural induction on R, with a case analysis according to its form.

$R = R'\backslash a$: we have $(\!|(R'\backslash a)@m|\!) = (\!|R'@m|\!)\backslash a$ with $a \notin m$.

By inductive hypothesis $(\!|R'@m|\!) = \mathtt{nf}(\mathtt{H}_m[(\!|R'|\!)])$, hence $(\!|R'@m|\!)\backslash a = \mathtt{nf}(\mathtt{H}_m[(\!|R'|\!)])\backslash a$. Thanks to the definition of the encoding $(\!|\cdot|\!)$ and of normal form, we have:

$$\mathtt{nf}(\mathtt{H}_m[(\!|R'|\!)])\backslash a = \mathtt{nf}(\mathtt{H}_m[(\!|R'|\!)\backslash a]) = \mathtt{nf}(\mathtt{H}_m[(\!|R'\backslash a|\!)]) = \mathtt{nf}(\mathtt{H}_m[(\!|R|\!)])$$

as desired.

$R = C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m') \triangleright P_l]$: by applying encoding $(\!|\cdot|\!)$ on R, we have:

$$(\!|R|\!) = (\!|C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m') \triangleright P_l]|\!) = \mathtt{H}_{m'}[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]]$$

hence:

$$\begin{aligned}(\!|R@m|\!) =& (\!|C_{l\in L}[l \mapsto m_l@(\langle\uparrow\rangle \cdot m') \triangleright P_l]@m|\!) = \\ & \mathtt{H}_{m'@m}[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]] = \\ & \mathtt{H}_m[\mathtt{H}_{m'}[C_{l\in L}[l \mapsto (\!|\delta(m_l) \triangleright P_l|\!)]]] = \mathtt{H}_m[(\!|R|\!)]\end{aligned}$$

where we used Lemma 9. Note that $\mathtt{H}_m[(\!|R|\!)] = \mathtt{nf}(\mathtt{H}_m[(\!|R|\!)])$ since $\mathtt{H}_m$ and $\mathtt{H}_{m'}$ do not contain restrictions, the top operator of context C is a parallel composition, and the encoding of monitored processes produces CCSK processes already in normal form. Hence, the thesis follows.

$R = m' \triangleright P$: by applying encoding $(\!|\cdot|\!)$ on R, we have $(\!|m' \triangleright P|\!) = \mathtt{H}_{m'}[P]$, hence

$$(\!|(m' \triangleright P)@m|\!) = \mathtt{H}_{m'@m}[P] = \mathtt{H}_m[\mathtt{H}_{m'}[P]] = \mathtt{H}_m[(\!|R|\!)].$$

Note that $\mathtt{H}_m[(\!|R|\!)] = \mathtt{nf}(\mathtt{H}_m[(\!|R|\!)])$ since $\mathtt{H}_m$ and $\mathtt{H}_{m'}$ do not contain restrictions, and P is standard. Hence, the thesis follows.

Item (2) follows immediately from the definition of the encoding, noting that $\mathtt{H}_{\langle\rangle} = \bullet$. □

Having all the necessary properties, we can prove the following theorem stating that encoding of the CCSK process X into RCCS, and encoding it back to CCSK is equal up to α-conversion to the normal form of the process X.

Theorem 6 *Let X be a reachable CCSK process. Then* $(\!|[\![X, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X)$.

Proof The proof is by structural induction on X, with a case analysis on the form of X. We first consider the case of standard processes, that is $X = P$.

$X = P$**:** by using the definitions of encodings $[\![\cdot]\!]$ and $(\!|\cdot|\!)$ we have:

$$(\!|[\![P, \langle\rangle]\!]|\!) = (\!|\langle\rangle \triangleright P|\!) = \mathtt{H}_{\langle\rangle}[P] = P = \mathtt{nf}(P)$$

as desired.

We now consider non-standard processes.

$X = \alpha[k].Y + \sum_{j\in J} \alpha_j.P_j$**:** by using the definition of encoding $[\![\cdot]\!]$, we have:

$$(\!|[\![\alpha[k].Y + \sum_{j\in J} \alpha_j.P_j, \langle\rangle]\!]|\!) = (\!|[\![Y, \langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot \langle\rangle]\!]|\!)$$

Thanks to Lemma 3 we have:

$$(\!|[\![Y, \langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot \langle\rangle]\!]|\!) = (\!|[\![Y, \langle\rangle]\!]@\langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot \langle\rangle|\!)$$

By Lemma 13 we have:

$$(\!|[\![Y, \langle\rangle]\!]@\langle k, \alpha, \sum_{j\in J} \alpha_j.P_j\rangle \cdot \langle\rangle|\!) = \mathtt{nf}(\mathtt{H}_{\langle k,\alpha,\sum_{j\in J} \alpha_j.P_j\rangle}[(\!|[\![Y, \langle\rangle]\!]|\!)])$$

Thanks to the inductive hypothesis we have $(\!|[\![Y, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(Y)$, and hence:

$$\begin{aligned}\mathtt{nf}(\mathtt{H}_{\langle k,\alpha,\sum_{j\in J} \alpha_j.P_j\rangle}[(\!|[\![Y, \langle\rangle]\!]|\!)]) =_\alpha & \mathtt{nf}(\mathtt{H}_{\langle k,\alpha,\sum_{j\in J} \alpha_j.P_j\rangle}[\mathtt{nf}(Y)]) = \\ & \mathtt{nf}(\alpha[k].\mathtt{nf}(Y) + \sum_{j\in J} \alpha_j.P_j) = \\ & \mathtt{nf}(\alpha[k].Y + \sum_{j\in J} \alpha_j.P_j)\end{aligned}$$

as desired.

$X = X_1 \mid X_2$**:** by definition of the encoding$[\![\cdot]\!]$, we have:

$$(\!|[\![X_1 \mid X_2, \langle\rangle]\!]|\!) = (\!|[\![X_1, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![X_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]|\!)$$

Thanks to Lemma 3 we can rewrite $[\![X_1, \langle\uparrow\rangle \cdot \langle\rangle]\!]$ and $[\![X_2, \langle\uparrow\rangle \cdot \langle\rangle]\!]$ as:

$$\begin{array}{rcl}
[\![X_1, \langle\uparrow\rangle \cdot \langle\rangle]\!] & = & C_{i\in I}^{[\![X_1,\langle\rangle]\!]}[i \mapsto m_i@(\langle\uparrow\rangle \cdot \langle\rangle) \triangleright P_i] = \\
& & C_{i\in I}^{[\![X_1,\langle\rangle]\!]}[i \mapsto m_i \triangleright P_i]@(\langle\uparrow\rangle \cdot \langle\rangle) = \\
& & [\![X_1, \langle\rangle]\!]@(\langle\uparrow\rangle \cdot \langle\rangle) \\
[\![X_2, \langle\uparrow\rangle \cdot \langle\rangle]\!] & = & C_{j\in J}^{[\![X_2,\langle\rangle]\!]}[j \mapsto m_j@(\langle\uparrow\rangle \cdot \langle\rangle) \triangleright P_j] = \\
& & C_{j\in J}^{[\![X_2,\langle\rangle]\!]}[j \mapsto m_j \triangleright P_j]@(\langle\uparrow\rangle \cdot \langle\rangle) = \\
& & [\![X_2, \langle\rangle]\!]@(\langle\uparrow\rangle \cdot \langle\rangle)
\end{array}$$

and obtain:

$$(\!|[\![X_1 \mid X_2, \langle\rangle]\!]|\!) = (\!|[\![X_1, \langle\rangle]\!]@(\langle\uparrow\rangle \cdot \langle\rangle) \mid [\![X_2, \langle\rangle]\!]@(\langle\uparrow\rangle \cdot \langle\rangle)|\!)$$

Thanks to Lemma 13 we have:

$$(\!|[\![X_1 \mid X_2, \langle\rangle]\!]|\!) = (\!|[\![X_1, \langle\rangle]\!]|\!) \mid (\!|[\![X_2, \langle\rangle]\!]|\!))$$

By inductive hypothesis we have that $(\!|[\![X_1, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X_1)$ and $(\!|[\![X_2, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X_2)$. Moreover, by definition of the normal form we have that:

$$\mathtt{nf}(X_1 \mid X_2) = \mathtt{nf}(X_1) \mid \mathtt{nf}(X_2)$$

Then we can conclude by noticing that:

$$(\!|[\![X_1, \langle\rangle]\!]|\!) \mid (\!|[\![X_2, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X_1) \mid \mathtt{nf}(X_2) = \mathtt{nf}(X_1 \mid X_2) = \mathtt{nf}(X)$$

as desired.

$X = X_1 \backslash a$**:** by using the definition of encodings $[\![\cdot]\!]$ and $(\!|\cdot|\!)$, we have:

$$(\!|[\![X_1 \backslash a, \langle\rangle]\!]|\!) = (\!|[\![X_1, \langle\rangle]\!] \backslash a|\!) = (\!|[\![X_1, \langle\rangle]\!]|\!) \backslash a$$

By inductive hypothesis we have $(\!|[\![X_1, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X_1)$ and we can conclude by noticing that:

$$(\!|[\![X_1, \langle\rangle]\!]|\!) \backslash a =_\alpha \mathtt{nf}(X_1) \backslash a = \mathtt{nf}(X_1 \backslash a) = \mathtt{nf}(X)$$

□

We can now prove operational correspondence results for the encoding.

Theorem 7 (Forward Correctness and Completeness) *Let R and S be two reachable RCCS processes. There exists an RCCS transition $R \xrightarrow{k,\alpha} S$ iff there exists a CCSK transition $(\!|R|\!) \xrightarrow{\alpha[k]} (\!|S'|\!)$ with $S \equiv S'$.*

Proof **($\Rightarrow$)** Since from Theorem 5 we have $R = [\![(\!|R|\!), \langle\rangle]\!]$ and $S = [\![(\!|S|\!), \langle\rangle]\!]$, we get $[\![(\!|R|\!), \langle\rangle]\!] \xrightarrow{k,\alpha} [\![(\!|S|\!), \langle\rangle]\!]$, then by Theorem 3 we have that $(\!|R|\!) \xrightarrow{\alpha[k]} (\!|S'|\!)$ with $S' \equiv S$.

($\Leftarrow$) If $(\!|R|\!) \xrightarrow{\alpha[k]} (\!|S'|\!)$ with $S' \equiv S$, then by Theorem 1 we have that $[\![(\!|R|\!), \langle\rangle]\!] \xrightarrow{k,\alpha} [\![(\!|S'|\!), \langle\rangle]\!]$. Then from Theorem 5 we have $R \xrightarrow{k,\alpha} S'$. The thesis follows by applying RCCS rule R-EQUIV. □

Theorem 8 (Backward Correctness and Completeness) *Let R and S be two reachable RCCS processes. There exists an RCCS transition $R \overset{k,\alpha}{\rightsquigarrow} S$ iff there exists a CCSK transition $(\!|R|\!) \overset{\alpha[k]}{\rightsquigarrow} (\!|S'|\!)$ with $S \equiv S'$.*

Proof The proof is similar to the one of Theorem 7. □

3.3 Isomorphism

In this Section we show that the LTS of RCCS up to structural congruence and the LTS of CCSK up to α-conversion and normal form, are isomorphic. The proof relies on the operational correspondence results for two encodings that we presented in the Sections 3.1 and 3.2. After that we argue about the *uniform* encoding.

We start with introducing the notion of isomorphism defined between two LTSs and the concept of the LTS up to equivalence relation.

Definition 20 (LTS Isomorphism) *Two LTSs $LTS_i : \langle \mathcal{P}_i, \mathcal{L}_i, \rightarrow_i \rangle$, with $i \in \{1, 2\}$ are* isomorphic *iff there exist two bijective functions $\gamma_L : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ and $\gamma_P : \mathcal{P}_1 \rightarrow \mathcal{P}_2$ such that $P_1 \xrightarrow{\alpha} P_2$ iff $\gamma_P(P_1) \xrightarrow{\gamma_L(\alpha)} \gamma_P(P_2)$.*

Definition 21 (LTS up to equivalence) *Given an $LTS : \langle \mathcal{P}, \mathcal{L}, \rightarrow \rangle$ and an equivalence relation $\sim$ on $\mathcal{P}$, we denote by $[P]$ the equivalence class of $P \in \mathcal{P}$. We define the LTS up to $\sim$ as $LTS_\sim : \langle \mathcal{P}_\sim, \mathcal{L}, \rightarrow_\sim \rangle$ where $\mathcal{P}_\sim$ is the set of equivalence classes of $\mathcal{P}$ modulo $\sim$, and $[P_1] \xrightarrow{l}_\sim [P_2]$ iff there exist $P_1 \in [P_1]$ and $P_2 \in [P_2]$ such that $P_1 \xrightarrow{l} P_2$.*

Before stating the main theorem, we need a few auxiliary lemmas.

Lemma 14 *For each CCSK processes X and Y, $[\![X, \langle\rangle]\!] \equiv [\![Y, \langle\rangle]\!]$ implies $[\![X, \langle\rangle]\!] = [\![Y, \langle\rangle]\!]$.*

Proof RCCS structural congruence just allows one to split threads with a toplevel parallel composition, and to put restrictions which are at toplevel inside threads outside the same threads. One can easily notice that in the encoding threads correspond to standard processes, hence parallel threads are split and restrictions moved outside iff the corresponding processes are not standard. The thesis follows. □

On CCSK processes, equivalence relation $\overset{ccsk}{\sim}$, is defined as follows: $X \overset{ccsk}{\sim} Y$ iff $\mathtt{nf}(X) =_\alpha \mathtt{nf}(Y)$. If we take RCCS processes we consider structural congruence as equivalence relation.

Lemma 15 *Let X and Y be CCSK processes. If $X \overset{ccsk}{\sim} Y$ then $[\![X, \langle\rangle]\!] =_\alpha [\![Y, \langle\rangle]\!]$.*

Proof By definition of $\overset{ccsk}{\sim}$ we have $\mathtt{nf}(X) =_\alpha \mathtt{nf}(Y)$. It is trivial to see that encoding CCSK processes equal up to α-conversion gives RCCS processes equivalent up to α-conversion. One can also notice that the encoding moves all the restrictions in the non-standard part outside the corresponding process, hence the thesis follows. □

With the next theorem, we will prove our main result showing that forward LTSs of RCCS and CCSK are isomorphic. The same is holding for the backward ones.

Theorem 9 (RCCS and CCSK are isomorphic) *The forward LTS of RCCS up to structural congruence is isomorphic to the forward LTS of CCSK up to $\overset{ccsk}{\sim}$. Both LTSs are restricted to reachable processes. The same result holds for backward LTSs.*

Proof We will show that thesis holds for the forward LTSs. The thesis for backward LTSs follows by using the Loop Lemma for CCSK (Lemma 2) and RCCS (Lemma 1).

The function on labels mapping $\alpha[k]$ to k, α is trivially bijective. The function on states maps an equivalence class of CCSK processes $[X]$ into $[[\![X, \langle\rangle]\!]]$. We remark that the function is well defined thanks to Lemma 15. In order to show that it is bijective we show that the function $(\!|\cdot|\!)$ is its inverse. It is easy to see that also function $(\!|\cdot|\!)$ is well defined on equivalence classes since structural congruence just allows one to move restrictions and parallel composition between monitored processes and processes, but this difference is immaterial in CCSK. Also, both calculi have α-conversion. Given an RCCS process R we have $[\![(\!|R|\!), \langle\rangle]\!] = R$ from Theorem 5. Given a CCSK process X we have $(\!|[\![X, \langle\rangle]\!]|\!) =_\alpha \mathtt{nf}(X)$ from Theorem 6. This proves bijectivity.

We need to show that $[X_1] \xrightarrow{\alpha[k]}_{\underset{\sim}{ccsk}} [X_2]$ iff $[[\![X_1, \langle\rangle]\!]] \xrightarrow{k,\alpha}_{\equiv} [[\![X_2, \langle\rangle]\!]]$. By expanding the definitions of LTS up to equivalence, we need to show that there exist X_1' and X_2' such that $X_1' \xrightarrow{\alpha[k]} X_2'$ with $\mathtt{nf}(X_1') =_\alpha \mathtt{nf}(X_1)$ and $\mathtt{nf}(X_2') =_\alpha \mathtt{nf}(X_2)$ iff there exist S_1 and S_2 such that $[\![X_1, \langle\rangle]\!] \equiv S_1 \xrightarrow{k,\alpha} S_2 \equiv [\![X_2, \langle\rangle]\!]$. By applying RCCS rule R-EQUIV the second part can be equivalently written as $[\![X_1, \langle\rangle]\!] \xrightarrow{k,\alpha} [\![X_2, \langle\rangle]\!]$.

Let us show the implication from left to right. Using Lemma 12 we obtain $\mathtt{nf}(X_1') \xrightarrow{\alpha[k]} X_2''$ with $\mathtt{nf}(X_2'') = \mathtt{nf}(X_2')$. Using again Lemma 12 and α-conversion we get $X_1 \xrightarrow{\alpha[k]} X_2'''$ with $\mathtt{nf}(X_2''') =_\alpha \mathtt{nf}(X_2'')$. As a result we also have $\mathtt{nf}(X_2''') =_\alpha \mathtt{nf}(X_2')$. From Theorem 1 we have $[\![X_1, \langle\rangle]\!] \xrightarrow{k,\alpha} [\![X_2''', \langle\rangle]\!]$. From Lemma 15 we have $[\![X_2''', \langle\rangle]\!] =_\alpha [\![X_2', \langle\rangle]\!]$ as desired.

Let us show the implication from right to left. From $[\![X_1, \langle\rangle]\!] \xrightarrow{k,\alpha} [\![X_2, \langle\rangle]\!]$ using Theorem 3 we have that $X_1 \xrightarrow{\alpha[k]} X_2'$ in CCSK, with $[\![X_2, \langle\rangle]\!] \equiv [\![X_2', \langle\rangle]\!]$. From Lemma 14 we have that $[\![X_2, \langle\rangle]\!] = [\![X_2', \langle\rangle]\!]$. The thesis follows. $\square$

We would like to remark that proving isomorphism between two calculi, implies that they can be equated by any behavioural equivalence,

including back and forth bisimilarity used in [58]. Even the result that we have shown is very strong, it is obtained via encodings which are not uniform [62]. We recall below the notion of uniform encoding.

Definition 22 (Uniform Encoding) *An encoding* $(\!|\cdot|\!)$ *is* uniform *iff it is:*

***homomorphic w.r.t. the parallel composition operator*:** *given two processes,* $\mathtt{R}_1$ *and* $\mathtt{R}_2$, $(\!|\mathtt{R}_1 \mid \mathtt{R}_2|\!) = (\!|\mathtt{R}_1|\!) \mid (\!|\mathtt{R}_2|\!)$,

***renaming preserving*:** *given a process* $\mathtt{R}$ *and an injective renaming function* σ *mapping names to names and keys to keys,* $(\!|\mathtt{R}\sigma|\!) = (\!|\mathtt{R}|\!)\sigma$.

The definition above differs from the Definition 13 for the fact that we require renamings to map keys to keys and names to names. The reason for it, is that in CCSK and RCCS, keys and names belongs to different sets and should not be mixed.

With the next proposition we show that our encoding of CCSK into RCCS is not uniform.

Proposition 1 *The encoding* $[\![\bullet]\!]$ *of CCSK into RCCS is not uniform.*

Proof It is easy to see that the encoding is not homomorphic w.r.t. parallel composition, since:

$$[\![a \mid b]\!] = [\![a \mid b, \langle\rangle]\!] = [\![a, \langle\uparrow\rangle \cdot \langle\rangle]\!] \mid [\![b, \langle\uparrow\rangle \cdot \langle\rangle]\!] = \langle\uparrow\rangle \cdot \langle\rangle \triangleright a \mid \langle\uparrow\rangle \cdot \langle\rangle \triangleright b$$

is not equal to:

$$[\![a]\!] \mid [\![b]\!] = [\![a, \langle\rangle]\!] \mid [\![b, \langle\rangle]\!] = \langle\rangle \triangleright a \mid \langle\rangle \triangleright b$$

Notably, this last process is not reachable in RCCS. □

With the next proposition we show that reachable processes in RCCS are not closed under parallel composition. Hence, the fact that the encoding is not uniform is not surprising.

Proposition 2 *Given two reachable RCCS processes* R *and* S*, there is no context* $C[1 \mapsto \cdot, 2 \mapsto \cdot]$ *such that* $C[1 \mapsto R, 2 \mapsto S]$ *is reachable.*

Proof Consider a reachable process R. It is easy to show by induction on the derivation from the initial process to R that the following invariant holds: each memory m inside R contains a number of $\langle\uparrow\rangle$ elements equal to the number of parallel composition operators in the path from the root to m. Given that C adds at least one parallel composition operator in the path to R and to S, if the invariant holds for R and S it cannot hold for their parallel composition. Note in fact that additional $\langle\uparrow\rangle$ elements need to be added inside memories, hence cannot be part of the C context. $\square$

Consequence of the proposition above is that encoding from CCSK to RCCS that map parallel operator of CCSK into parallel operator of RCCS (what was required for uniform encoding) does not exist. Moreover, there is no encoding that can map the CCSK parallel operator to any RCCS context.

Considering encoding from RCCS to CCSK, it is not possible to state an uniformity result. Since reachable RCCS processes are not closed under parallel composition, the terms in the statement would not be all reachable.

3.4 Cross-fertilisation Results

In this Section we shall show some cross-fertilization results by using the theory developed in the previous sections.

Namely, we bring the *Reverse Diamond Property* from CCSK into RCCS, and in opposite direction, we import *Parabolic Lemma* of RCCS into CCSK. As addition to that, we add the irreversible actions to CCSK following the guidance in [23; 43]. We would like to remark that CCSK Parabolic Lemma has been proved in the literature [70, Lemma 5.12], yet the direct proof is more complex than importing the result from RCCS.

The Reverse Diamond Property [70, Proposition 5.10], stated below, shows that backward transitions are confluent.

Property 2 (CCSK Reverse Diamond Property) *Let X, Y and Z be reachable CCSK processes.*

1. *if $X \overset{\alpha[k]}{\rightsquigarrow} Y$ and $X \overset{\beta[k]}{\rightsquigarrow} Z$ then $\alpha = \beta$ and $Y = Z$.*

2. *if* $X \overset{\alpha[k]}{\rightsquigarrow} Y$ *and* $X \overset{\beta[h]}{\rightsquigarrow} Z$ *with* $k \neq h$ *then there exists* W *such that* $Y \overset{\beta[h]}{\rightsquigarrow} W$ *and* $Z \overset{\alpha[k]}{\rightsquigarrow} W$.

By bringing the result above to the RCCS, we obtain following property:

Property 3 (RCCS Reverse Diamond Property) *Let* R, R_1 *and* R_2 *be reachable RCCS processes.*

1. *if* $R \overset{k,\alpha}{\rightsquigarrow} R_1$ *and* $R \overset{k,\beta}{\rightsquigarrow} R_2$ *then* $\alpha = \beta$ *and* $R_1 \equiv R_2$.

2. *if* $R \overset{k,\alpha}{\rightsquigarrow} R_1$ *and* $R \overset{h,\beta}{\rightsquigarrow} R_2$ *with* $k \neq h$ *then there exists* S *such that* $R_1 \overset{h,\beta}{\rightsquigarrow} S$ *and* $R_2 \overset{k,\alpha}{\rightsquigarrow} S$.

Proof We have two cases, one for each item in the statement.

1. By hypothesis we have that $R \overset{k,\alpha}{\rightsquigarrow} R_1$ and $R \overset{k,\beta}{\rightsquigarrow} R_2$. By applying Theorem 8 we have that $(\!|R|\!) \overset{\alpha[k]}{\rightsquigarrow} (\!|R'_1|\!)$ with $R'_1 \equiv R_1$ and $(\!|R|\!) \overset{\beta[k]}{\rightsquigarrow} (\!|R'_2|\!)$ with $R'_2 \equiv R_2$. By Reverse Diamond Property on CCSK transitions (Property 2) we have that $(\!|R'_1|\!) = (\!|R'_2|\!)$ and $\alpha = \beta$. By Theorem 5 we have that $[\![(\!|R'_1|\!), \langle\rangle]\!] = R'_1$ and $[\![(\!|R'_2|\!), \langle\rangle]\!] = R'_2$, with $R'_1 = R'_2$. Since $R_1 \equiv R'_1$ and $R'_2 \equiv R'_2$ we also have $R_1 \equiv R_2$ as desired.

2. By hypothesis we have that $R \overset{k,\alpha}{\rightsquigarrow} R_1$ and $R \overset{h,\beta}{\rightsquigarrow} R_2$ with $k \neq h$. By applying Theorem 8 we have that $(\!|R|\!) \overset{\alpha[k]}{\rightsquigarrow} (\!|R'_1|\!)$ with $R'_1 \equiv R_1$ and $(\!|R|\!) \overset{\beta[h]}{\rightsquigarrow} (\!|R'_2|\!)$ with $R'_2 \equiv R_2$. Since, $k \neq h$ by Reverse Diamond Property on CCSK transitions (Property 2) we have that there exists W such that $(\!|R'_1|\!) \overset{\beta[h]}{\rightsquigarrow} W$ and $(\!|R'_2|\!) \overset{\alpha[k]}{\rightsquigarrow} W$. Let S be a RCCS process such that $(\!|S|\!) = W$. By Theorem 8 we have that $(\!|R'_1|\!) \overset{\beta[h]}{\rightsquigarrow} (\!|S|\!)$ implies $R'_1 \overset{h,\beta}{\rightsquigarrow} S'$ with $S' \equiv S$, and $(\!|R'_2|\!) \overset{\alpha[k]}{\rightsquigarrow} (\!|S|\!)$ implies $R'_2 \overset{k,\alpha}{\rightsquigarrow} S''$ with $S'' \equiv S$. Since $R_1 \equiv R'_1$, $S' \equiv S$, $R_2 \equiv R'_2$ and

$S'' \equiv S$, by applying RCCS rule R-EQUIV we have that $R_1 \overset{h,\beta}{\rightsquigarrow} S$ and $R_2 \overset{k,\alpha}{\rightsquigarrow} S$, as desired. $\square$

Reversible computation can be decomposed into a backward only computation followed by forward only computation. This is stated in Parabolic Lemma [23, Lemma 10]. Intuitively, this means that the process can first go backward so to enable as many choices as possible, and then go only forward.

Proposition 3 (RCCS Parabolic Lemma) *For any reachable RCCS process R, if $R \rightleftharpoons \cdots \rightleftharpoons S$, then there exists R' such that $R \rightsquigarrow^* R' \rightarrow^* S$.*

The proof of the Parabolic Lemma for CCSK by exploiting the isomorphism of two reversible calculi is shown with a following proposition.

Proposition 4 (CCSK Parabolic Lemma) *For any reachable CCSK process X, if $X \rightleftharpoons \cdots \rightleftharpoons Y$, then there exists X' such that $X \rightsquigarrow^* X' \rightarrow^* Y$.*

Proof By hypothesis we have that $X \rightleftharpoons \cdots \rightleftharpoons Y$. Let $R = [\![X, \langle\rangle]\!]$ and $S = [\![Y, \langle\rangle]\!]$. We can apply Theorem 1 to each forward transition and Theorem 2 to each backward transition to obtain a sequence of transitions $R = [\![X, \langle\rangle]\!] \rightleftharpoons \cdots \rightleftharpoons [\![Y, \langle\rangle]\!] = S$. We can then apply Proposition 3 to the sequence of transitions above and obtain that there exists R' such that $R \rightsquigarrow^* R' \rightarrow^* S$. Since $R \rightsquigarrow^* R'$ with $R = [\![X, \langle\rangle]\!]$ by applying Theorem 4 (and rule R-EQUIV) we have that $X \rightsquigarrow^* X'$ with $[\![X', \langle\rangle]\!] \equiv R'$. From the reduction $R' \rightarrow^* S$, by applying Theorem 3 (and rule R-EQUIV) we obtain that $X' \rightarrow^* Y$ with $[\![Y, \langle\rangle]\!] \equiv S$, as desired. $\square$

Irreversible actions in CCSK. Irreversible or commit actions are the actions which cannot be backtracked. When executing an irreversible action, all the actions executed before on the same term sequentially, are blocked and unable to be backtracked as well. For instance, consider the process $a.b.P$, where b is the commit action. After the execution of the actions a and b, we have process $a[k].b[h].P$, where action b cannot be undone. As a consequence, action a is blocked and cannot be reversed as

$$\text{(CCSK Processes)} \quad X, Y ::= \mathbf{0} \mid \sum_{i \in I} \pi_i.X_i \mid (X \mid Y) \mid X \backslash a$$

$$\text{(CCSK Prefixes)} \quad \pi ::= \alpha \mid \alpha[k] \mid \underline{\alpha} \mid \underline{\alpha}[*]$$

Figure 10: CCSK with irreversible actions syntax

well. Following the approach given in [23; 43], irreversible actions can be added into CCSK that we defined in Section 2.2.

The set of the actions $\mathtt{Act}$ is divided into two disjunctive sets A and A^i, representing sets of reversible and irreversible actions, respectively. We let μ, λ to range over set $\mathtt{Act}_\tau = A_\tau \cup A^i_\tau$, while α, β will represent only reversible actions ($\alpha, \beta \in A_\tau = A \cup \{\tau\}$) and $\underline{\alpha}, \underline{\beta}$ only irreversible ones ($\underline{\alpha}, \underline{\beta} \in A^i_{\underline{\tau}} = A^i \cup \{\underline{\tau}\}$).

The syntax of CCSK with irreversible actions is given in Figure 10. The only difference in respect to CCSK is in the prefix definition. An irreversible action $\underline{\alpha}$ can be annotated only with element $*$ that will tell us that executed action cannot be reversed. In this case, there is no need of recording the key of the executed action, since after its execution, there is no possible backward step on the same prefix.

Semantics is given in Figure 11. The only difference is in the rule KI-COMMIT and KI-SYN. By applying rule KI-COMMIT, action $\underline{\alpha}$ is annotated with the $*$ that will behave as a tag, telling us that this action cannot be reverted. The obtained process can continue execution in the forward direction, but it is not able to go backward. The difference between rules KI-SYN1 and KI-SYN2 is that first one allows synchronisation of the reversible actions and the second one is considering commit actions. The backward rules are symmetric to the forward one, with rule KI-COMMIT being an exception. There is no backward rule for commit action. To have intuition about semantics for commit actions, we give the following example:

Example 11 Let us consider the process $P = a.\underline{c}.P \mid b.\underline{\bar{c}}$. By executing the actions $a[k]$ and $b[h]$, we obtain the CCSK process $X = a[k].\underline{c}.P \mid b[h].\underline{\bar{c}}$.

(KI-ACT1)

$$\alpha.P \xrightarrow{\alpha[k]} \alpha[k].P$$

(KI-ACT2)

$$\frac{X \xrightarrow{\lambda[h]} X' \qquad k \neq h}{\alpha[k].X \xrightarrow{\lambda[h]} \alpha[k].X'}$$

(KI-COMMIT)

$$\underline{\alpha}.P \xrightarrow{\underline{\alpha}[k]} \underline{\alpha}[*].P$$

(KI-SUM)

$$\frac{X_j \xrightarrow{\lambda[k]} X'_j \ \forall i \neq j. X_i = X'_i \wedge \mathtt{std}(X_i)}{\sum_{i \in I} X_i \xrightarrow{\lambda[k]} \sum_{i \in I} X'_i}$$

(KI-PAR-L)

$$\frac{X \xrightarrow{\lambda[k]} X' \qquad \mathtt{fresh}(k, Y)}{X \mid Y \xrightarrow{\lambda[k]} X' \mid Y}$$

(KI-PAR-R)

$$\frac{Y \xrightarrow{\lambda[k]} Y' \qquad \mathtt{fresh}(k, X)}{X \mid Y \xrightarrow{\lambda[k]} X \mid Y'}$$

(KI-SYN1)

$$\frac{X \xrightarrow{\alpha[k]} X' \qquad Y \xrightarrow{\bar{\alpha}[k]} Y' \qquad \alpha \neq \tau}{X \mid Y \xrightarrow{\tau[k]} X' \mid Y'}$$

(KI-SYN2)

$$\frac{X \xrightarrow{\underline{\alpha}[k]} X' \qquad Y \xrightarrow{\bar{\underline{\alpha}}[k]} Y' \qquad \underline{\alpha} \neq \underline{\tau}}{X \mid Y \xrightarrow{\underline{\tau}[k]} X' \mid Y'}$$

(KI-RES)

$$\frac{X \xrightarrow{\lambda[k]} X' \qquad \lambda \notin \{a, \bar{a}, \underline{a}, \bar{\underline{a}}\}}{X \backslash a \xrightarrow{\lambda[k]} X' \backslash a}$$

Figure 11: CCSK with irreversible actions forward semantics

By applying rule KI-SYN with the premises $a[k].\underline{c}.P \xrightarrow{\underline{c}[w]} a[k].\underline{c}[*].P$ and $b[h].\bar{\underline{c}} \xrightarrow{\bar{\underline{c}}[w]} b[h].\bar{\underline{c}}[*]$, we obtain:

$$X = a[k].\underline{c}.P \mid b[h].\bar{\underline{c}} \xrightarrow{\underline{\tau}[w]} a[k].\underline{c}[*].P \mid b[h].\bar{\underline{c}}[*] = Y$$

The only possible computation for the process Y is to continue forward execution by executing the forward actions of the process P. There is no possible backward execution since processes $a[k].\underline{c}[*].P$ and $b[h].\bar{\underline{c}}[*]$ are blocked with element [*].

Chapter 4

Causality notions in π-calculus (background knowledge)

In this Section we recall syntax and semantics of the π-calculus [73], and use it as a base for its three well-known causal extensions.

In π-calculus is the process calculus that allows one to naturally express processes which can dynamically change their structure. The important property of π-calculus is name mobility, i.e. capability of sending names along channels where received names can be used as channel names in the future communications. Process can send (extrude) private name to the environment and make it free for the future computations.

Let assume an existence of two mutually disjoint infinite sets: the set of names $\mathcal{N}$, ranged over a, b, c and the set of variables $\mathcal{V}$, ranged over x, y.

Syntax of π-calculus is given in Figure 12. Processes are given by productions P, Q. Idle process is represented with $\mathbf{0}$. A prefixed process is written as $\pi.P$, where prefix can be: an output $\overline{b}a$, where name a is sent over a channel b, an input $b(x)$, where a certain name will be received over a channel b and bound to variable x or the silent action τ. In the prefixes $\overline{b}a$ and $b(x)$, name b is used in *subject* position, while name a and variable

x are in the *object* position. To get the subject and object of a prefix, we use operators $sub(\cdot)$ and $obj(\cdot)$. The above definitions and operators are extended to actions as well Parallel composition of processes P and Q is represented with $P \mid Q$, while $\nu a(P)$ represents that name a is restricted in P. The action α can be: the same as prefix π (an input, an output and the τ-action) or bound output $\overline{b}\langle\nu a\rangle$ in which bound name a is sent over a channel b. We let α to range over set of all actions, denoted with $\mathtt{Act}$.

In π-calculus we have two binder construct: name a bound via restriction operator in the process P, written as $\nu a(P)$, and variable x bound in input prefixed process $b(x).P$. In following, we define the functions that compute sets of bound names and variables.

Definition 23 (Set of bound names/variables) *The sets of bound names and variables of the given process P, written as $\mathtt{bn}(P)$ and $\mathtt{bv}(P)$, respectively, are defined as:*

$$\begin{array}{ll}
\mathtt{bn}(P \mid Q) = \mathtt{bn}(P) \cup \mathtt{bn}(Q) & \mathtt{bv}(P \mid Q) = \mathtt{bv}(P) \cup \mathtt{bv}(Q) \\
\mathtt{bn}(\nu a(P)) = \mathtt{bn}(P) \cup \{a\} & \mathtt{bv}(\nu a(P)) = \mathtt{bv}(P) \\
\mathtt{bn}(\overline{b}a.P) = \mathtt{bn}(P) & \mathtt{bv}(\overline{b}a.P) = \mathtt{bv}(P) \\
\mathtt{bn}(a(x).P) = \mathtt{bn}(P) & \mathtt{bv}(a(x).P) = \mathtt{bv}(P) \cup \{x\}
\end{array}$$

Sets of the free names and variables of the process P are defined as $\mathtt{fn}(P) = \mathtt{n}(P) \setminus \mathtt{bn}(P)$ and $\mathtt{fv}(P) = \mathtt{v}(P) \setminus \mathtt{bv}(P)$ where $\mathtt{n}(P)$ and $\mathtt{v}(P)$ denote the sets of all names and variables appearing in process P. Notation above is extended to the actions as well:

$$\begin{array}{ll}
\mathtt{bn}(\overline{b}a) = \emptyset & \mathtt{bv}(\overline{b}a) = \emptyset \\
\mathtt{bn}(a(x)) = \emptyset & \mathtt{bv}(a(x)) = \{x\} \\
\mathtt{bn}(\overline{b}\langle\nu a\rangle) = \{a\} & \mathtt{bv}(\overline{b}\langle\nu a\rangle) = \emptyset
\end{array}$$

Given a process P, we represent the set of its bound names and variables, written $\mathtt{bound}(P)$ as $\mathtt{bound}(P) = \mathtt{bn}(P) \cup \mathtt{bv}(P)$. Similarly, set of the free names and variables is written as $\mathtt{free}(P)$.

Notation 3 To make notation uniform through all semantics concerning π-calculus, we write:

- an output, an input and bound output actions as $\overline{b}a$, $b(x)$ and $\overline{b}\langle\nu a\rangle$, respectively. (As they are defined for π-calculus in Figure 12)

$$
\begin{array}{ll}
\text{(Processes)} & P, Q ::= \mathbf{0} \mid \pi.P \mid (P \mid Q) \mid \nu a(P) \\
\text{(Prefixes)} & \pi ::= \overline{b}a \mid b(x) \mid \tau \\
\text{(Actions)} & \alpha ::= \pi \mid \overline{b}\langle \nu a \rangle
\end{array}
$$

Figure 12: π-calculus syntax

- P to specify that process is classic π-calculus process.

Now we give the operational semantics for the π-calculus.

Definition 24 (π-calculus Semantics) *The operational semantics of π-calculus is defined as LTS $(\mathcal{P}, \rightarrow, \texttt{Act})$ where $\mathcal{P}$ is the set of π-calculus processes and* `Act` *is the set of actions. Transition relations $\rightarrow$ is the smallest relations induced by the rules in Figure 13.*

Late semantics for the π-calculus is given in Figure 13. Process $\pi.P$ can perform an action π through the rule ACT. The action is transformed into a label and the computation continues with the process P. Rule PAR-L allows left component in the parallel composition to execute the action α provided that the bound names of α are not free in the process in parallel Q. Similar for the rule PAR-R. Two processes can communicate through rules COM and CLOSE where necessary substitution is applied on the variable x. Rule RES allows restricted process to perform the action α if restricted name is not one of the names in α. Process can extrude (i.e. send name to the environment and make it free for the future computations) a restricted name over a rule OPEN.

In order to have a better intuition about semantics of the π-calculus, we give the following example.

Example 12 Consider the process $P = \nu a(\overline{b}a \mid \overline{c}a \mid a(x))$ where name a is restricted in the process P. To execute the action $a(x)$ we first need to 'open' the name a. For instance, it can be done by action $\overline{c}\langle \nu a \rangle$ where rule OPEN is applied:

$$\nu a(\overline{b}a \mid \overline{c}a \mid a(x)) \xrightarrow{\overline{c}\langle \nu a \rangle} \overline{b}a \mid a(x)$$

$$\text{(ACT)}\quad \pi.P \xrightarrow{\pi} P \qquad\qquad \text{(PAR-L)}\quad \frac{P \xrightarrow{\alpha} P' \qquad \mathtt{bound}(\alpha) \cap \mathtt{free}(Q) = \emptyset}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

$$\text{(PAR-R)}\quad \frac{Q \xrightarrow{\alpha} Q' \qquad \mathtt{bound}(\alpha) \cap \mathtt{free}(P) = \emptyset}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \qquad\qquad \text{(COM)}\quad \frac{P \xrightarrow{\overline{b}a} P' \qquad Q \xrightarrow{b(x)} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'\{^a/_x\}}$$

$$\text{(CLOSE)}\quad \frac{P \xrightarrow{\overline{b}\langle \nu a\rangle} P' \qquad Q \xrightarrow{b(x)} Q'}{P \mid Q \xrightarrow{\tau} \nu a(P' \mid Q'\{^a/_x\})} \qquad \text{(RES)}\quad \frac{P \xrightarrow{\alpha} P' \qquad a \notin \mathtt{n}(\alpha)}{\nu a(P) \xrightarrow{\alpha} \nu a(P')} \qquad \text{(OPEN)}\quad \frac{P \xrightarrow{\overline{b}a} P' \qquad a \neq b}{\nu a(P) \xrightarrow{\overline{b}\langle \nu a\rangle} P'}$$

Figure 13: π-calculus semantics

In the resulting process $\overline{b}a \mid a(x)$ name a is free and now, both actions can be executed. Let us perform first the output on the channel b and then the action $a(x)$:

$$\overline{b}a \mid a(x) \xrightarrow{\overline{b}a} a(x) \xrightarrow{a(x)} \mathbf{0}$$

In the following, we recall some notions on transitions in π-calculus (transitions are represented as $t : P \xrightarrow{\alpha} Q$).

Definition 25 *Given two transitions t_1 and t_2, they are:*

- coinitial *if they have the same source, i.e. $t_1 : P \xrightarrow{\alpha} P_1$ and $t_2 : P \xrightarrow{\beta} P_2$;*
- cofinal *if they have the same target, i.e. $t_1 : P_1 \xrightarrow{\alpha} Q$ and $t_2 : P_2 \xrightarrow{\beta} Q$;*
- composable *if target of one is source of the other transition, i.e. $t_1 : P \xrightarrow{\alpha} Q$ and $t_2 : Q \xrightarrow{\beta} Q_1$;*

A trace *is a sequence of pairwise composable transitions, written as $t_1; t_2$. The* empty *trace is denoted with ϵ.*

Causality in π-calculus. Causality in π-calculus was studied in [14], where authors defined two forms of causal dependencies: subject (struc-

tural) and the object (link) dependency. Subject dependency in the π-calculus is determined by nesting of the prefixes. For instance, in the process $\overline{b}a.\overline{c}d$, the action $\overline{c}d$ structurally depend on the action $\overline{b}a$. The object dependency is induced by extrusion of a name. For example, in the process $\nu a(\overline{b}a \mid a(x))$, the output action $\overline{b}a$ needs to be executed before the input action on the channel a. In that way, after the execution of the bound output $\overline{b}\langle\nu a\rangle$, name a is free. The interesting problem while defining object causality for the π-calculus is parallel extrusion of the same name. For instance, in the process $\nu a(\overline{b}a \mid \overline{b}a \mid a(x))$ if both extrusions are executed before the action $a(x)$ (as in Example 12), there are different interpretations about which of the actions $\overline{b}a$ and $\overline{c}a$ actually causes the input action. Therefore, diverse definition of the object dependency yield to different approaches to causality in π-calculus.

In this document we will consider three different approaches to causality in π-calculus, and the main causal semantics representing them [14; 19; 20], are given in Sections 4.1, 4.2 and 4.3, respectively.

4.1 Causal semantics by Boreale and Sangiorgi

In this Section we revise a compositional causal semantics for standard π-calculus (forward only) introduced in [14].

Originally the causal semantics was defined for a polyadic π-calculus, with early semantics for inputs. Here we adapt the causal semantics to work with monadic π-calculus and late input semantics.

The authors distinguish between two forms of dependencies: subject and the object. While the object dependence can be detected from the trace (run) that process did, to track the subject one, authors introduced a causal term, defined on the top of π-calculus [1]. Every visible transition is bound with unique cause $k \in \mathcal{K}$, where $\mathcal{K}$ is a set of causes. Let $\mathcal{N}$ and $\mathcal{V}$ be the infinitive, countable sets of names and variables such that $\mathcal{N} \cap \mathcal{V} \cap \mathcal{K} = \emptyset$.

[1] We use the π-calculus which syntax and semantics are defined in Figures 12 and 13

The syntax of the causal process is defined as follows:

$$\text{(Causal process) } A, B ::= P \mid \mathtt{K} :: A \mid A \mid B \mid \nu a(A)$$

where P is a π-calculus process. In causal term $\mathtt{K} :: A$, set $\mathtt{K}$ records that every action performed by A depends on $\mathtt{K}$. Two causal processes can be composed in parallel by $A \mid B$ and name a can be restricted in the process A. Set of the causes appearing in the causal process A is denoted with $\mathcal{K}(A)$ and its definition is given bellow.

Definition 26 (Set of the causes) *The set of the causes of a given causal process A, written as $\mathcal{K}(A)$, is inductively defined on the structure of the causal term as:*

$$\mathcal{K}(A \mid B) = \mathcal{K}(A) \cup \mathcal{K}(B) \qquad \mathcal{K}(\mathtt{K} :: A) = \mathtt{K} \cup \mathcal{K}(A)$$
$$\mathcal{K}(\nu a(A)) = \mathcal{K}(A) \qquad \mathcal{K}(P) = \emptyset$$

In what follows, we give the definition of the labels on the transitions and operational semantics of the causal term.

Definition 27 (Label on the transition) *Label on the transition of the causal process is defined as $A \xrightarrow[\mathtt{K}]{k:\alpha} A'$, where cause set $\mathtt{K}$ contains causes of all the actions that trigger the action α, k is the cause associated to α and $\alpha ::= \overline{b}a \mid b(x) \mid \overline{b}\langle \nu a \rangle \mid \tau$.*

Rules for causal semantics are given in Figure 14. Causes are introduced into the processes by the rules BS-OUT and BS-IN. A new cause k is attached to the executing action. Rule BS-CAU allows a causal process $\mathtt{K}' :: A$ to move if A can move, while cause set $\mathtt{K}'$ is preserved. Rules BS-OPEN and BS-RES are defined in usual way. The communication between two causal processes can be done through rules BS-COM and BS-CLOSE, while necessary substitution is applied. Notation $A_1'[k \rightsquigarrow \mathtt{K}_2]$ indicates the fact that cause k needs to be replaced with the set $\mathtt{K}_2$. The condition $k \notin \mathcal{K}(A_1, A_2)$ ensures that cause k is fresh. The synchronisation rules, merge the cause sets of processes that communicate and do not produce a new cause bound to τ. The reason for this is because τ action do not impose causes to future actions. In order to have a better intuition of how semantics work, we give the following example.

Example 13 Let us consider the π-calculus process $P = \overline{a}b.\overline{c}d.\mathbf{0} \mid \overline{e}a_1.c(x).Q$. Process P can perform the output on the channel a, and we have:

$$\overline{a}b.\overline{c}d.\mathbf{0} \mid \overline{e}a_1.c(x).Q \xrightarrow[\emptyset]{k:\overline{a}b} \{k\} :: \overline{c}d.\mathbf{0} \mid \overline{e}a_1.c(x).Q = A$$

We can notice that cause k is bound to the action $\overline{a}b$, and saved in the resulting process. Since P was π-calculus process, the cause set $\mathtt{K}$ is empty. To continue execution, process A can perform the output on the channel e:

$$k :: \overline{c}d.\mathbf{0} \mid \overline{e}a_1.c(x).Q \xrightarrow[\emptyset]{k_1:\overline{e}a_1} \{k\} :: \overline{c}d.\mathbf{0} \mid \{k_1\} :: c(x).Q$$

The two actions executed do not depend structurally and this is the reason why cause set is empty in both cases. Now we can synchronise two processes in parallel:

$$k :: \overline{c}d.\mathbf{0} \mid k_1 :: c(x).Q \xrightarrow{\tau} \{k, k_1\} :: \mathbf{0} \mid \{k_1, k\} :: Q\{^d/_x\}$$

As we can notice, after the communication, cause sets $\{k\}$ and $\{k_1\}$ are merged meaning that the actions of Q will structurally depend on both actions $\overline{a}b$ and $\overline{e}a_1$.

Remark 5 In the original version of the causal semantics [14], labels on transitions are defined as $A \xrightarrow[\mathtt{K};\, k]{\alpha} A'$. We use the notation $A \xrightarrow[\mathtt{K}]{k:\alpha} A'$ to simplify the comparison given in Section 5.6. For the same reason we divided original rule for communication COM from [14] into two rules: BS-COM and BS-CLOSE.

The subject causality is given by cause sets attached to the process, while the object one is defined on the trace that process did. The first action that extrudes a bound name causes every further action using that name in subject or object position of the label. To illustrate this, we give the following example.

Example 14 Let us consider a process $P = \nu a(\overline{b}a \mid \overline{c}a \mid a(x))$ and the trace:

$$\nu a(\overline{b}a \mid \overline{c}a \mid a(x)) \xrightarrow[\emptyset]{k:\overline{b}\langle\nu a\rangle} A_1 \xrightarrow[\emptyset]{k_1:\overline{c}a} A_2 \xrightarrow[\emptyset]{k_2:a(x)} \{k\} :: \mathbf{0} \mid \{k_1\} :: \mathbf{0} \mid \{k_2\} :: \mathbf{0}$$

where $A_1 = \{k\} :: \mathbf{0} \mid \overline{c}a \mid a(x)$ and $A_2 = \{k\} :: \mathbf{0} \mid \{k_1\} :: \mathbf{0} \mid a(x)$. The action $\overline{b}\langle\nu a\rangle$ extrudes name a and cause other two actions: $\overline{c}a$ where name a is in the object position and $a(x)$ where a is in the subject position.

$$\text{(BS-OUT)}\ \overline{b}a.A \xrightarrow[\emptyset]{k:\overline{b}a} k :: A \qquad \text{(BS-IN)}\ b(x).A \xrightarrow[\emptyset]{k:b(x)} k :: A$$

$$\text{(BS-CAU)}\ \frac{A \xrightarrow[\mathtt{K}]{k:\alpha} A'}{\mathtt{K}' :: A \xrightarrow[\mathtt{K},\mathtt{K}']{k:\alpha} \mathtt{K}' :: A'}$$

$$\text{(BS-PAR)}\ \frac{A_1 \xrightarrow[\mathtt{K}]{k:\alpha} A_1' \quad \mathtt{bound}(\alpha) \cap \mathtt{free}(A_2) \neq \emptyset}{A_1 \mid A_2 \xrightarrow[\mathtt{K}]{k:\alpha} A_1' \mid A_2}$$

$$\text{(BS-COM)}\ \frac{A_1 \xrightarrow[\mathtt{K}_1]{k:\overline{b}a} A_1' \quad A_2 \xrightarrow[\mathtt{K}_2]{k:b(x)} A_2' \quad k \notin \mathcal{K}(A_1, A_2)}{A_1 \mid A_2 \xrightarrow{\tau} A_1'[k \rightsquigarrow \mathtt{K}_2] \mid A_2'\{^a/_x\}[k \rightsquigarrow \mathtt{K}_1]}$$

$$\text{(BS-OPEN)}\ \frac{A \xrightarrow[\mathtt{K}]{k:\overline{b}a} A'}{\nu a\ A \xrightarrow[\mathtt{K}]{k:\overline{b}\langle \nu a \rangle} A'} \qquad \text{(BS-RES)}\ \frac{A \xrightarrow[\mathtt{K}]{k:\alpha} A' \quad a \notin \mathtt{n}(\alpha)}{\nu a\ A \xrightarrow[\mathtt{K}]{k:\alpha} \nu a\ A'}$$

$$\text{(BS-CLOSE)}\ \frac{A_1 \xrightarrow[\mathtt{K}_1]{k:\overline{b}\langle \nu a \rangle} A_1' \quad A_2 \xrightarrow[\mathtt{K}_2]{k:b(x)} A_2' \quad k \notin \mathcal{K}(A_1, A_2)}{A_1 \mid A_2 \xrightarrow{\tau} \nu a(A_1'[k \rightsquigarrow \mathtt{K}_2] \mid A_2'\{^a/_x\}[k \rightsquigarrow \mathtt{K}_1])}$$

Figure 14: Causal semantics rules

We adapt the definition of object causality to the late semantics defined on the traces of the causal process A.

Definition 28 (object causality) *In a trace* $A_1 \xrightarrow[K_1]{k_1:\alpha_1} A_2 \cdots A_n \xrightarrow[K_n]{k_n:\alpha_n} A_{n+1}$ *where* A_1 *is a* π*-calculus process* P*, if*

- $\alpha_i = \overline{b}\langle \nu a \rangle$ *where* $a \cap \mathtt{fn}(A_i) = \emptyset$ *and for all* $j < i$*,* $a \cap \mathtt{n}(\alpha_j) = \emptyset$ *we say that name* a *is* introduced *in* α_i*. Action* α_h *is* object dependent *on* α_i*,* $1 \leqslant i < h \leqslant n$*, if there is a name introduced in* α_i *which is among the free names of* α_h*.*
- $\alpha_i = b(x)$ *where* $x \cap \mathtt{fn}(A_i) = \emptyset$ *and for all* $j < i$*,* $x \cap \mathtt{v}(\alpha_j) = \emptyset$ *we say that variable* x *is* introduced *in* α_i*. Action* α_h *is* object dependent *on*

α_i, $1 \leqslant i < h \leqslant n$, if there is a variable introduced in α_i which is among the free variables of α_h.

4.2 Causal semantics by Crafa, Varacca and Yoshida

The authors introduced a compositional event structure semantics for the forward π-calculus [19]. They represent a process as a pair $(\mathbf{E}, \mathbf{X})$, where $\mathbf{E}$ is a prime event structure and $\mathbf{X}$ is a set of bound names. Structural causality is encoded by causal relation of $\mathbf{E}$, while the object one is defined through the notion of *permitted configuration* (Definition 31). In the following, we recall definition of the prime event structure, given in [19].

Definition 29 (Labelled Prime Event Structures) *A labelled prime event structure is a tuple $\mathbf{E} = \langle E, \leq, \smile, \lambda \rangle$ such that*

- *E is a set of events ranged over e;*
- *$\langle E, \leq \rangle$ represents the partial and it is also called* causal order*;*
- *for every $e \in E$,* enabling set *of e, $[e) := \{e' | e' < e\}$ is finite;*
- conflict relation *$\smile$ is an irreflexive and symmetric relation for which holds: for every $e_1, e_2, e_3 \in E$ if $e_1 \leq e_2$ and $e_1 \smile e_3$ then $e_2 \smile e_3$;*
- *labelling function $\lambda : E \to L$, where L is a set of labels, associates a label to every event in E*

The actions from the π-calculus are represented with labels, while events depict the occurrences of the actions. With labels, we are able to identify events representing different occurrences of the same name. The computation in event structure is captured by configurations which represent the trace of the event structure in which events are causally ordered.

The semantics of the π-calculus (given in Figure 12) represented with prime event structures, is defined in Figure 15. Idle process is represented with empty pair. Prefixed process $\pi.P$ is defined as event structure $\pi.\mathbf{E}_P$, where prefix operation adds a minimal element π bellow every element in the event structure $\mathbf{E}_P$ (the event structure that represents the process P). The set of the bound names $\mathbf{X}_P$ is the same as one of process P (i.e.

$$
\begin{array}{lcl}
\mathbf{0} & = & (\emptyset, \emptyset) \\
\pi.P & = & (\pi.\mathbf{E}_P, \mathbf{X}_P) \\
\nu a(P) & = & (\mathbf{E}_P, \mathbf{X}_P \cup \{a\}) \\
P \mid Q & = & (\mathbf{E}_P \parallel_\pi \mathbf{E}_Q, \mathbf{X}_P \cup \mathbf{X}_Q)
\end{array}
$$

Figure 15: Event structure semantics for π-calculus

$\mathbf{X}_P = \mathtt{fn}(P)$). Restriction operator adds restricted name a into a set of bound names $\mathbf{X}_P$, while the event structure is the same as for process P. Parallel composition of two processes merge sets of bound names for processes P and Q, while the resulting event structure is defined through the parallel composition operator for event structures $\parallel_\pi$, defined as categorical product followed by relabelling and restriction:

$$\mathbf{E}_1 \parallel_\pi \mathbf{E}_2 = ((\mathbf{E}_1 \times \mathbf{E}_2)[f_\pi][er])\backslash\{\mathtt{bad}\}$$

where restriction operation on the event structures $\mathbf{E}\backslash a$, removes from $\mathbf{E}$ element labelled with a and events that are above it (i.e. events that are causally dependent on a) and relabelling $\mathbf{E}[f]$ compose function λ of $\mathbf{E}$ with f, where $f : L \to L'$ and L and L' are two sets of labels. Labelling function of a new event structure is $f \circ \lambda$.

Intuitively, relabelling function f_π is defined to relabel: every pair $(a(x), \overline{a}b)$ into $\tau_{x\to b}$, symbolising that during the synchronisation, name b substitutes variable x; pairs like $(y(x), \overline{a}b)$ into $(y(x), \overline{a}b)_{x\to b}$ representing that decision if the synchronisation is correct (if action $\alpha_{y\to b}$ in which variable y is substituted with name a happened before, then synchronisation will be correct) is postponed; each pair $(a(x), \overline{c}b)$, where channels are different names, into bad (meaning that synchronisation is not allowed).

Example 15 Let as consider the process $P_3 = P_1 \mid P_2$ where $P_1 = a(x).\overline{c}d$ and $P_2 = \overline{a}b$. The event structures $\mathbf{E_1}, \mathbf{E_2}$ and $\mathbf{E_3}$ given in Figure 15, represent processes P_1, P_2 and P_3, respectively. Causal order is represented with the straight lines, while conflict is represented with dotted lines. From the event structure $\mathbf{E}_3$ we can conclude that we have two possibilities for the computation: or action $a(x)$ will execute and trigger

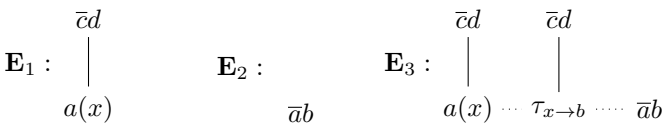


Figure 16: The event structures representing processes P_1, P_2 and P_3

the action $\overline{c}d$, while in parallel input action $\overline{a}b$ can be performed; or processes will synchronise and then trigger the action $\overline{c}d$. We can notice that synchronisation pair $(a(x), \overline{a}b)$ is relabelled into $\tau_{x\to b}$.

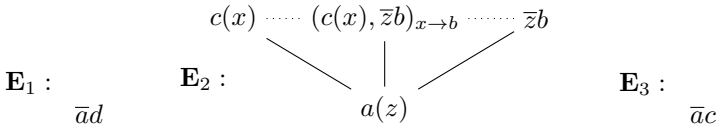
The relabelling function f_π is formally defined as follows.

Definition 30 (Relabelling function) *Let $L = \{b(x), \overline{b}a, \tau\}$ be the set of the labels in π-calculus without bound output, and let $L' = L \cup \{(\alpha, \beta)_{x\to b} | \alpha, \beta \in L\} \cup \{\tau_{x\to b}, \texttt{bad}\}$ be extended set of labels, where* `bad` *is a distinguished label. With $Pom(L')$ are denoted partially ordered multisets of L' (pomsets). The relabelling function $f_\pi : Pom(L') \times (L' \uplus \{*\} \times L' \uplus \{*\}) \to L'$ is defined as follows:*

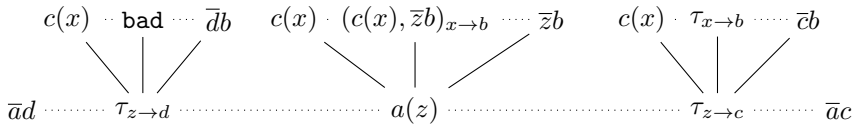
$$
\begin{aligned}
&f_\pi(X, \langle a(y), \overline{a}b\rangle) = \tau_{y\to b}\\
&f_\pi(X, \langle a(y), \overline{c}b\rangle) = \texttt{bad}\\
&f_\pi(X, \langle a(x), \overline{y}c\rangle) = \begin{cases} \tau_{x\to c} & \textit{if } \alpha_{y\to a} \in X\\ (a(x), \overline{y}c)_{x\to c} & \textit{otherwise}\end{cases}\\
&f_\pi(X, \langle y(x), \overline{a}b\rangle) = \begin{cases} \tau_{x\to b} & \textit{if } \alpha_{y\to a} \in X\\ (y(x), \overline{a}b)_{x\to b} & \textit{otherwise}\end{cases}\\
&f_\pi(X, \langle y(x), *\rangle) = \begin{cases} a(x) & \textit{if } \alpha_{y\to a} \in X\\ y(x) & \textit{otherwise}\end{cases}\\
&f_\pi(X, \langle \overline{y}b, *\rangle) = \begin{cases} \overline{a}b & \textit{if } \alpha_{y\to a} \in X\\ \overline{y}b & \textit{otherwise}\end{cases}\\
&f_\pi(X, \langle \alpha, *\rangle) = \alpha\\
&f_\pi(X, \langle \alpha, \beta\rangle) = \texttt{bad}
\end{aligned}
$$

After synchronisation has been completed, the information carried by the τ actions is not needed anymore. For that reason, second relabelling function er is defined to delete the subscript of the τ actions. In order to have a better intuition about the semantics, we show one more example.

Example 16 Let us consider process $P = P_1 \mid P_2 \mid P_3$ where $P_1 = \overline{a}d$, $P_2 = a(z).(c(x) \mid \overline{z}b)$ and $P_3 = \overline{a}c$. The event structures representing processes P_1, P_2 and P_3 are given bellow:



The pair $(c(x), \overline{z}b)$ is relabelled as $(c(x), \overline{z}b)_{x \to b}$ representing the fact that synchronisation will happen if the input action $a(z)$ receive the name c. The event structure representing the process P is shown bellow



There are two possibilities for the synchronisation, pairs $(a(z), \overline{a}d)$ and $(a(z), \overline{a}c)$ and this is represented by $\tau_{z \to d}$ and $\tau_{z \to c}$. If the communication $\tau_{z \to d}$ happen, further synchronisation pair $(c(x), \overline{z}d)$ is relabelled to bad (generally, event bad is omitted). If synchronisation $\tau_{z \to c}$, it allows further communication on the channel c and $\tau_{x \to b}$ is obtained.

To represent *disjunctive (object) causality*, the authors use so-called the inclusive approach, in which, to execute an action with a bound subject it is required that at least one extruder of the bound name was executed previously, but it is not necessary to record which one. To model it, they introduce the notion of *permitted configurations*, where a configuration is permitted if beside the action with a bound subject, contains the action that extruded the bound name. Formally:

Definition 31 (Permitted configurations) *Let* $(\mathbf{E}, \mathbf{X})$ *be an event structure where* $\mathbf{X}$ *is a set of bound names and* $e \uparrow = \{e' | e \leq e'\}$ *for some event* $e \in E$*. A configuration* C *of* $\mathbf{E}$ *is* permitted *in* $(\mathbf{E}, \mathbf{X})$ *whenever for any event* $e \in C$ *which label has in the subject position* $a \in \mathbf{X}$*,*

- $C \backslash e \uparrow$ *is permitted and*
- $C \backslash e \uparrow$ *contains an event which label is an output action with name* a *in the object position of the label*

Definition 32 (Subject and Object causality) *Let the event structure semantics of the* π*-calculus process* P*, be* $P = (\mathbf{E}_P, \mathbf{X}_P)$ *and events* $e_1, e_2 \in E_P$*, then*

- e_2 *is* subject dependent *on* e_1 *if* $e_1 \leq_{\mathbf{E}_P} e_2$*;*
- e_2 *is* object dependent *on* e_1 *if the label of* e_1 *is the output of the name* $a \in \mathbf{X}_P$ *where name* a *is also the subject of the label of* e_2*; and exists a configuration* C *permitted in* $(\mathbf{E}_P, \mathbf{X}_P)$*, where* $e_1, e_2 \in C$*.*

To have an intuition about the causality and permitted configuration, we give the following example.

Example 17 Let us consider the process $P = \nu a(\overline{b}a \mid \overline{c}a.d(y) \mid a(x))$, The semantics of the process π represented with the event structures is $P = \mathbf{E}_P, \mathbf{X}_P)$, where $\mathbf{X}_P = \{a\}$. From Definition 32 action $d(y)$ is structurally dependent on the action $\overline{c}a$. Action $a(x)$ is object dependent on the actions $\overline{b}a$ and $\overline{c}a$, since permitted configurations are $C_1 = \{\overline{b}a, a(x)\}$ and $C_2 = \{\overline{c}a, a(x)\}$.

Definition 33 (Concurrent Events) *Two events are* concurrent *if they are not causally related and not in conflict.*

The object causality defined above ensures that at least one of the extruders precedes action with a bound subject. To track exact extruder, it is possible by duplicating action with a bound subject and letting every copy to depend on different extruders. For example, process $P = \nu a(\overline{b}a \mid \overline{c}a \mid a(x))$ is represented as:

$$\begin{array}{ccc} a(x) & \cdots & a(x) \\ | & & | \\ \overline{b}a & & \overline{c}a \end{array}$$

where actions $a(x)$ are in conflict: or $a(x)$ will depend on action $\overline{b}a$ or $a(x)$ will depend on action $\overline{c}a$.

4.3 Causal semantics for reversible π-calculus by Cristescu, Krivine and Varacca

In [20], authors presented a compositional semantics for the reversible π-calculus ($R\pi$). Reversibility is obtained by extending approach of RCCS to the π-calculus. History information is saved in the memories (organised as stacks of events) attached to every π-calculus process.

Let $\mathcal{I}$ be the set of action identifiers ranged over i, j, k with a special symbol $* \in \mathcal{I}$ representing a partial synchronisation. We let $\Delta,\ \Gamma$ to range over subsets of $\mathcal{I}$.

Syntax of $R\pi$ processes is given in Figure 17. Reversible processes R, S are built on the top of standard π-calculus processes (we use π-calculus defined in Figure 12). As for RCCS, **0** is the idle process and term $m \triangleright P$ is called monitored process, where m is a memory and P is a standard π-calculus process. Parallel composition of two monitored processes is represented with $R \mid S$. The restriction $\nu a_\Gamma(R)$, annotated with set Γ (initially empty), will behave as a classic π-calculus restriction when $\Gamma = \emptyset$, otherwise it represents the memory in which information about the extrusions of the name a are kept. Memories are organised as a stacks of events with the event representing the last action that process did, on the top of the stack (left in textual representation). Events $\langle\rangle$ and $\langle\uparrow\rangle$ symbolise the empty memory and split of the parallel composition, respectively. Memory event $\langle i, k, \pi\rangle$ contains identifier i, contextual cause k and event label π of the executed action. If executed action was output or bound output, event label is $\overline{b}a$; if process performed input action without synchronisation, the event label is $b[*/x]$ and it stands for partial synchronisation; if communication happened and name a is received over channel b, then it is written as $b[a/x]$. We let e to range over memory events. To have access to the identifier, contextual cause and label of the even e, we write $id(e), c(e)$ and $l(e)$, respectively. The actions are represented with α. Difference with π-calculus is that in the bound output

($R\pi$ processes)	$R,\ S ::= \mathbf{0} \mid m \triangleright P \mid (R \mid S) \mid \nu a_\Gamma(R)$
(Memories)	$m ::= \langle\rangle \mid \langle i, k, \pi\rangle.m \mid \langle\uparrow\rangle.m$
(Event labels)	$\pi ::= \overline{b}a \mid b[*/x] \mid b[a/x]$
(Actions)	$\alpha ::= \overline{b}a \mid b(x) \mid \overline{b}\langle\nu a_\Gamma\rangle \mid \tau$

Figure 17: $R\pi$ syntax

action, bound name a is annotated with the set Γ.

Labels on the transitions are defined as follows:

Definition 34 *The label ζ of a transition $t : R \xrightarrow{\zeta} S$ has a form:*

$$\zeta ::= (i, j, k) : \gamma \quad \textit{with} \quad \gamma ::= \alpha \mid \alpha^-$$

where $i \in \mathcal{I}\backslash\{*\}$, $j, k \in \mathcal{I}$ are identifier, instantiator and contextual cause of the transition t. Label α^- stands for the negative (backward) transition.

Substitutions in $R\pi$ are not executed directly but simply logged in event labels. Indeed the calculus uses what is called explicit substitution, which is implemented by looking up into the memories. This is why processes need to search in their memories for the public name of a channel in order to check that a synchronisation is possible.

Definition 35 *For all process of the form $m \triangleright \beta.P$, where prefix $\beta = b(x)$ or $\beta = \overline{b}a$, public label of β, $m[\beta]$, is defined as follows:*

$$\begin{array}{ll}
\langle\rangle[a] = a & (\langle i, k, b[*/x]\rangle.m)[x] = x \\
m[b(x)] = m[b](x) & (\langle\uparrow\rangle.m)[a] = m[a] \\
m[\overline{b}a] = \overline{m[b]}m[a] & (e.m)[a] = m[a] \\
(\langle i, k, b[c/x]\rangle.m)[x] = c &
\end{array}$$

In the following we define two ways to update the memory: synchronisation update (substitution) and contextual cause update.

Definition 36 *The* synchronisation update, *written as $R_{[a/x]@i}$ substitute a partial synchronisation label $[*/x]$ with a complete synchronisation label $[a/x]$*

in the event identified with i:

$$(R \mid S)_{[a/x]@i} = R_{[a/x]@i} \mid S_{[a/x]@i} \qquad (\nu b_\Gamma R)_{[a/x]@i} = \nu b_\Gamma (R)_{[a/x]@i}$$
$$(\langle i, _, b[*/x]\rangle.m \triangleright P)_{[a/x]@i} = \langle i, _, b[a/x]\rangle.m \triangleright P \qquad (m \triangleright P)_{[a/x]@i} = m \triangleright P$$

Definition can be extended to the contextual cause update, *written as $R_{[k/k']@i}$ that substitute old cause k' with a new one k in the event identified with i:*

$$(R \mid S)_{[k/k']@i} = R_{[k/k']@i} \mid S_{[k/k']@i} \qquad (\nu b_\Gamma R)_{[k/k']@i} = \nu b_\Gamma (R)_{[k/k']@i}$$
$$(\langle i, k', _\rangle.m \triangleright P)_{[k/k']@i} = \langle i, k, _\rangle.m \triangleright P \qquad (m \triangleright P)_{[k/k']@i} = m \triangleright P$$

Now we give the definitions of the three relations on the memory events that author considered.

Definition 37 (Relations on events) *Let R be a monitored process, the relations on the memory events of R are defined as follows:*

structural causal relation *: event e is structurally dependent on event e' in R, written as $e' \sqsubset_R e$ if exist $m \in R$ such that $m = m_2.e.m_1.e'.m_0$, for some m_2, m_1, m_0.*

contextual causal relation *: event e is contextually dependent on event e' in R, written as $e' \prec_R e$ if $c(e) = id(e')$.*

instantiation relation *: event e is instantiated by event e' in R, written as $e' \rightsquigarrow_R e$ if $e' \sqsubset_R e$ and $l(e') = b[a/x]$ for some a, b, x where x is in the subject position in $l(e)$. For all memories m, it is written $\texttt{inst}_m(x) = i$ if there is some event $\langle i, k, b[a/x]\rangle$ in m that instantiates x.*

If relations hold on events, then it will hold on identifiers and contextual causes as well. The same notation is used.

Example 18 Let us consider the process

$$R = \nu c_{\{i_2\}}(\langle i_1, *, \overline{a}b\rangle \triangleright P \mid \langle i_3, i_2, \overline{n}d\rangle \triangleright Q \mid \langle i_4, *, \overline{x}a\rangle.\langle i_2, *, \overline{e}n\rangle.\langle i_1, *, a[b/x]\rangle \triangleright \mathbf{0})$$

The relations between events are:

- $\langle i_1, *, a[b/x]\rangle \sqsubset_R \langle i_2, *, \overline{e}n\rangle$ event i_2 structurally depend on the event i_1
- $\langle i_2, *, \overline{e}n\rangle \prec_R \langle i_3, i_2, \overline{n}d\rangle$ event i_2 is contextual cause of the event i_3, since $c(i_3) = id(i_2)$

- $\langle i_1, *, a[b/x]\rangle \rightsquigarrow_R \langle i_4, *, \overline{x}a\rangle$ event i_1 instantiate the event i_4, since $\texttt{inst}_{\langle i_4,*,\overline{x}a\rangle.\langle i_2,*,\overline{e}n\rangle.\langle i_1,*,a[b/x]\rangle}(x) = i_1$

Operational semantics of $R\pi$ contains two groups of rules: positive, given in Figure 18 and negative, obtained from the positive by inversion ([20, Definition 2.4]).

$$\text{(IN)}\ \frac{i \notin m \quad j = inst_m(b)}{m \triangleright b(x).P \xrightarrow{(i,j,*):m[b(x)]} \langle i, *, b[*/x]\rangle.m \triangleright P}$$

$$\text{(OUT)}\ \frac{i \notin m \quad j = inst_m(b)}{m \triangleright \overline{b}a.P \xrightarrow{(i,j,*):m[\overline{b}a]} \langle i, *, \overline{b}a\rangle.m \triangleright P}$$

$$\text{(OPEN)}\ \frac{R \xrightarrow{(i,j,k):\alpha} R' \quad \alpha = \overline{b}a \vee \alpha = \overline{b}\langle \nu a_{\Gamma'}\rangle}{\nu a_\Gamma R \xrightarrow{(i,j,k):\overline{b}\langle \nu a_\Gamma\rangle} \nu a_{\Gamma+1} R'}$$

$$\text{(CAUSE REF)}\ \frac{R \xrightarrow{(i,j,k):\alpha} R' \quad a \in subj(\alpha) \wedge k = k' \text{ or } \exists k' \in \Gamma \quad k \rightsquigarrow_R k'}{\nu a_\Gamma R \xrightarrow{(i,j,k'):\alpha} \nu a_\Gamma R'_{[k'/k]@i}}$$

$$\text{(COM)}\ \frac{R \xrightarrow{(i,j,k):\overline{b}a} R' \quad S \xrightarrow{(i,j',k'):b(x)} S' \quad k =_* j' \wedge k' =_* j}{R \mid S \xrightarrow{(i,*,*):\tau} R' \mid S'_{[a/x]@i}}$$

$$\text{(CLOSE)}\ \frac{R \xrightarrow{(i,j,k):\overline{b}\langle \nu a_\Gamma\rangle} R' \quad S \xrightarrow{(i,j',k'):b(x)} S' \quad k =_* j' \wedge k' =_* j}{R \mid S \xrightarrow{(i,*,*):\tau} \nu a_\Gamma(R' \mid S'_{[a/x]@i})}$$

$$\text{(PAR)}\ \frac{R \xrightarrow{(i,j,k):\alpha} R'}{R \mid S \xrightarrow{(i,j,k):\alpha} R' \mid S} \qquad \text{(MEM)}\ \frac{R \equiv S \xrightarrow{\zeta} S' \equiv R'}{R \xrightarrow{\zeta} R'}$$

$$\text{(NEW)}\ \frac{R \xrightarrow{\zeta} R' \quad a \notin \zeta}{\nu a_\Gamma R \xrightarrow{\zeta} \nu a_\Gamma R'}$$

Figure 18: $R\pi$ forward semantics

$$
\begin{array}{ll}
(\textsc{Split}) & m \triangleright (P \mid Q) \equiv (\langle\uparrow\rangle.m \triangleright P \mid \langle\uparrow\rangle.m \triangleright Q) \\
(\textsc{Res}) & m \triangleright \nu a(P) \equiv \nu a_{\emptyset}(m \triangleright P) \text{ with } a \notin m
\end{array}
$$

Figure 19: $R\pi$ structural laws

Let us comment on the rules. Rules IN and OUT generate a fresh new identifier i bound to the performed action. New event e is added in the stack and attached to the memory m from the left in textual representation. Every time when rule OPEN is applied, the identifier of the performed action is added to the set Γ. The restriction νa_{Γ} is not discarded after the execution (as in standard π-calculus), but together with the identifier of the action, kept in the result process. In this way, information about extruders of the name a is preserved. Rule CAUSE REF is used to update the contextual cause (if it is necessary) when executing action is passing the memory νa_{Γ}. It can be applied only when $\Gamma \neq \emptyset$. The new cause k' can be the same as the old one k, or can be chosen from the set Γ under the condition that it was instantiate by old cause, i.e. $k \rightsquigarrow_R k'$. To communicate through the rules COM and CLOSE, two processes need to agree about the communication channel, identifier and to satisfy side condition which ensures that cause of the one process corresponds to the instantiator of the another process ($=_*$ is defined to be true if $k = j'$ or one of the elements is $*$). Additional condition in the rule CLOSE is that $a \notin \mathtt{fn}(S)$ whenever $\Gamma = \emptyset$. Rules PAR, MEM and NEW (rule for restriction) are defined as usual.

Structural rules are given in Figure 19. Parallel composition of processes sharing the memory m splits into two monitored process with the memory $\langle\uparrow\rangle.m$ (memory is duplicated and annotated with split event). Two resulting processes can continue with computing independently. Through the rule RES, restriction is pushed on the toplevel of monitored process with a condition that restricted name is not appearing in the memory of the process.

In order to have intuition of how semantics works, we show following

examples.

Example 19 Let us consider process $R = \nu b_\emptyset(\overline{a}b.c \mid d) \mid a(x).\bar{x}$. Process $\nu b_\emptyset(\overline{a}b.c \mid d)$ can extrude a private name b over channel a and then communicate with the rest of the process R through the rule CLOSE. We have:

$$\nu b_\emptyset(\overline{a}b.c \mid d) \mid a(x).\bar{x} \xrightarrow{(i,*,*):\tau} \nu b_\emptyset(\nu b_i(\langle i, *, \overline{a}b\rangle \triangleright c \mid d) \mid \langle i, *, a[b/x]\rangle \triangleright \bar{x})$$

Thanks to memory νb_i, we know that before the action i, the scope of the name b was process $\overline{a}b.c \mid d$.

Example 20 Let us consider a monitored process $R = \langle\rangle \triangleright \nu a_\emptyset(\overline{b}a \mid \overline{c}a \mid a(x))$ and actions to be executed, $\overline{c}a, \overline{b}a$, identified by i_1, i_2, respectively. The computation is presented bellow (the split events in the memories and empty memories are omitted):

$$\begin{aligned} R &\xrightarrow{(i_1,*,*):\overline{c}\langle\nu a_\emptyset\rangle} \nu a_{\{i_1\}}(\overline{b}a \mid \langle i_1, *, \overline{c}a\rangle \triangleright \mathbf{0} \mid a(x)) \\ &\xrightarrow{(i_2,*,*):\overline{b}\langle\nu a_{\{i_1\}}\rangle} \nu a_{\{i_1,i_2\}}(\langle i_2, *, \overline{b}a\rangle \triangleright \mathbf{0} \mid \langle i_1, *, \overline{c}a\rangle \triangleright \mathbf{0} \mid a(x)) = S \end{aligned}$$

Two actions are executed concurrently and both of them can be seen as extruders of the name a. Input action on the name a can pick its cause from the set $\Gamma = \{i_1, i_2\}$. Which action will be selected, depends on the context of the process, while in this case, we can chose randomly. By taking the output action on the channel b as a cause, we have:

$$S \xrightarrow{(i_3,*,i_2):a(x)} \nu a_{\{i_1,i_2\}}(\langle i_2, *, \overline{b}a\rangle \triangleright \mathbf{0} \mid \langle i_1, *, \overline{c}a\rangle \triangleright \mathbf{0} \mid \langle i_3, i_2, a[*/x]\rangle \triangleright \mathbf{0})$$

We can notice that identifier i_2 is saved in the memory event $\langle i_3, i_2, a[*/x]\rangle$, therefore, the action with identifier i_3 needs to be reversed before the one with identifier i_2. The event i_1 can be reversed at any time.

Definition 38 (Causality on the transitions) *Given a two transitions $t_1 : R \xrightarrow{(i_1,j_1,k_1):\gamma_1} S$ and $t_2 : S \xrightarrow{(i_2,j_2,k_2):\gamma_2} T$, where t_1 is not opposite of t_2, we have that:*

- *$t_1 \sqsubset t_2$, t_1 structurally cause transition t_2, if $i_1 \sqsubset_T i_2$ or $i_2 \sqsubset_R i_1$*
- *$t_1 \prec t_2$, t_1 contextually cause transition t_2, if $i_1 \prec_T i_2$ or $i_2 \prec_R i_1$*

Transition t_1 cause *transition t_2, written as $t_1 < t_2$ if $t_1 \sqsubset t_2$ or $t_1 \prec t_2$. Transitions are* concurrent *if they are not causally related.*

Chapter 5

Parametric Framework for Reversible π-Calculi

Three different semantics revised in Chapter 4 represent three different approaches to the problem of parallel extrusion of the same name. This problem depicts the essence of the object causality of the mentioned causal semantics. As an example, consider the process

$$\nu a(\overline{b}a \mid \overline{c}a \mid a(x))$$

where the discussion is on who will cause the input $a(x)$ In this work, we consider the following approaches to the problem above:

- The classical and most used approach is the one where the order of the actions that extrude name a is important and only the first of them is the cause of the action $a(x)$. There exist different causal semantics representing this idea [14; 17; 26] and all of them are defined for the standard π-calculus. We group together these semantics, since in [26] authors stated that after abstracting away from the models used to keep track of the causal dependences, the final order between the actions induced by their semantics coincides with the one given in [14; 17].

- Causal semantics introduced in [19] represents approach in which

action $a(x)$, from the example above, depends on one of the extruders, but there is no need to record on which one exactly. This semantics is defined for the standard (forward-only) π-calculus as well.

- The first compositional semantics for reversible π-calculus, introduced in [20], explains the problem above as: the action $a(x)$ depends on one of the extruders, which are executed concurrently. Exactly which one is decided by the context.

To study different approaches to causality in π-calculus (mentioned above) in the context of reversible computation, we need a model which can express different notions of causality. We start from the observation that object causality depends on the extruded names and how information about the extruder are stored. So the idea is to devise a framework for reversible π-calculi, parametric with respect to the data structure that stores information about the extrusions of a name. Different approaches to parallel extrusion problem can be obtained by using a different data structures. Additionally, the framework adds reversibility to the semantics that were defined only for the forward computations. Hence, it permits different orders of the causally-consistent backwards steps.

A preliminary discussion about causal semantics for π-calculus and initial idea about the framework is given [57]. We argued that it it was necessary to add causal information to the silent actions in the semantics [14].

This Chapter is based on our work introduced in [59], where we develop the idea behind the framework without adding causal information to the τ-actions. Here, we present a parametric framework for reversible π-calculi starting with an informal introduction of it, given in Section 5.1, followed by definitions of its syntax (Section 5.2) and operational semantics (Section 5.3). After that in Section 5.4, we define the data structures with operations on them and map observed causal semantics into the framework. We proved that reversibility in our framework is well defined by proving standard properties for reversible calculus, Loop Lemma, Square Lemma and causal consistency given in Section 5.5 (Lemma 18

,Lemma 20, Theorem 10 ,respectively). Additionally, in Section 5.6, we show causal correspondence between causal semantic given in [14] and corresponding instance of the framework. In the end of this Chapter (Section 5.7), we present the idea about causal bisimulation.

5.1 Informal presentation

General approach to devise a reversible extensions for the CCS-like calculi defined through SOS rules is given in [70]. The main ideas behind this approach are to make each operator of a calculus static and to use communication keys to uniquely identify the actions. While dynamic operators as choice or prefix are forgetful operators, in the static one, there is no loss of information. For instance, if we consider a CCS process $P = a.Q_1 \mid \overline{a}.Q_2$ the synchronisation between parallel components of the process P is:

$$a.Q_1 \mid \overline{a}.Q_2 \xrightarrow{\tau[i]} a[i].Q_1 \mid \overline{a}[i].Q_2$$

As we already pointed out in Section 2.2, prefixes a and $\overline{a}$ are not discarded, but annotated with a communication key i. Decorated prefixes are used only for the backward steps, hence resulting process can continue forward computing behaving as process $Q_1 \mid Q_2$. We expand this idea on the more complex setting, π-calculus, where possibility of creating new channel names and treating channels as sent values is enabled. For instance, by adapting the process P to the π-calculus, we have computation:

$$a(x).Q_1 \mid \overline{a}b.Q_2 \xrightarrow{i:\tau} a(x)[i].Q_1\{^{b^i}/_x\} \mid \overline{a}b[i].Q_2$$

In the substitution $\{^{b^i}/_x\}$, variable x is substituted by the name b decorated with the key i. Decorations on the names keep track of the substitutions occurring during the communications. In the example, it means that variable x is substituted with the name b in the synchronisation identified by the key i.

Remark 6 We could keep track about the substitution of the name differently. For example, we could record in the memory, beside the key i also the state of the process before the substitution, as it is done in [51].

Applying this approach to the π-calculus example above, we would have computation:

$$a(x).Q_1 \mid \overline{a}b.Q_2 \xrightarrow{i:\tau} a(x)[i, Q_1].Q_1\{^b/_x\} \mid \overline{a}b[i].Q_2$$

As we can notice the input prefix is annotated with both key and the state of the process Q_1 before the substitution. This approach is memory consuming, but allows one to keep track of the substitution while considering the calculi in which sent values can be processes as higher-order π-calculus [72].

In π-calculus, by addopting the static way of bookkeeping the past actions that process did as in [70], we avoid using the structural SPLIT rule of $R\pi$ [20]. As we mentioned in Section 4.3 (Figure 17), monitored process in $R\pi$, has the form $m \triangleright P$, where memory m keeps track of the past events of the process. One drawback of this approach is necessity of split rule to enable parallel composition of the processes sharing the same memory to perform the forward action. The structural rule of the form

$$m \triangleright (P \mid Q) \equiv \langle\uparrow\rangle \cdot m \triangleright P \mid \langle\uparrow\rangle \cdot m \triangleright Q$$

is not associative and, as shown in [51], brings undesired feature in which equivalent processes performing the same action may become non-equivalent processes.

Besides keeping track about the actions executed in the past of the process, framework has to remember the actions which extruded the certain name. Following the idea introduced in [20], this can be done by using the contextual cause of an action. For example in the computation

$$\nu a\,(\overline{b}a \mid a(x).P) \xrightarrow{i:\overline{b}\langle\nu a\rangle} \nu a_{\{i\}}(\overline{b}a[i] \mid a(x).P) \xrightarrow{j:a(x)} \nu a_{\{i\}}(\overline{b}a[i] \mid a(x)[j, i].P)$$

we can notice that after the extrusion of the name a (first action performed), the restriction νa is not discarded as in the standard π-calculus, but transformed into the memory $\nu a_{\{i\}}$. In this way, the fact that name a is extruded by the action i, is recorded. The memory $\nu a_{\{i\}}$ is not behaving as a restriction operator any more, hence name a is free and the input action on the channel a can be executed. The contextual cause of the action $a(x)$ is i and this is recorded in the process $a(x)[j, i].P$.

5.2 Syntax of the framework

In this Section we present the syntax of the framework.

We assume the existence of the denumerable infinite mutually disjoint sets: the set of names $\mathcal{N}$, the set of keys $\mathcal{K}$, and the set of variables $\mathcal{V}$, ranged over names a, b, c, keys i, j, k and variables x, y. We let $*$ to be a distinguished key such that $\mathcal{K}_* = \mathcal{K} \cup \{*\}$.

The syntax of the framework is presented in Figure 20. The standard π-calculus (we use definition of the π-calculus given in Figures 12 and 13) processes are given by P, Q productions. The idle process is represented with $\mathbf{0}$. The output prefixed process $\overline{b}a.P$ indicates the fact that name a can be sent over a channel b, while the input prefixed process $b(x).P$ symbolises that some name can be received and bound to variable x over a channel b. Parallel composition of two processes is represented with $P \mid Q$, while $\nu a(P)$ symbolises the fact that name a is restricted in P.

Reversible processes given by X, Y productions are defined on the top of the π-calculus processes. Differently from the standard π-calculus, performed actions are not discarded, but annotated and kept into the structure of a process, representing its *history*. A reversible process $\mathbf{P}$ behaves as a standard π-calculus process P, only decorated with instaniators. The instantiators are used to keep track of the substitutions. The prefix $\overline{b}^j a^{j_1}[i, K]$ is called a *past output* and it records the fact that in the past, process performed an output action identified by key i and that the contextual cause set of the executed action was $K \subseteq \mathcal{K}_*$. Prefix $b^j(x)[i, K].X$ represents a *past input* recording the fact that executed action was the input action identified by key i and its contextual cause set was K. Parallel composition of two reversible processes X and Y is represented with $X \mid Y$. Inspired by [20], the restriction operator νa_Δ is decorated with the memory Δ which keeps track of the extruders of the name a. When Δ is empty ($\texttt{empty}(\Delta) = true$, we will give the precise definition in the further text), νa_Δ will act as the classical restriction operator νa of the π-calculus. We denote the set of reversible processes with $\mathcal{X}$.

Notation 4 If the prefix of the process is not relevant, we will denote it with π. Hence, we have $\pi = \overline{b}^j a^{j_1}$ or $\pi = b^j(x)$. To specify that j is the

$$
\begin{array}{rl}
X,\ Y ::= & \mathbf{P} \mid \overline{b}^{j} a^{j_1}[i,K].X \ \mid\ b^{j}(x)[i,K].X \ \mid\ X \mid Y \ \mid \nu a_{\Delta}(X) \\
P,\ Q ::= & \mathbf{0} \mid \overline{b}a.P \mid b(x).P \mid\ P \mid Q \ \mid \nu a(P)
\end{array}
$$

Figure 20: Syntax of the framework

instantiator of one of the names in the prefix π, we will use notation $j \in \pi$ if $\pi = \overline{c}^{j} d^{j_1}$ or $\pi = \overline{c}^{j_1} d^{j}$ or $c^{j}(x)$ for some $c, d \in \mathcal{N}$ and $j_1 \in \mathcal{K}_*$. We shall use notation b^* in the subject or object position of the label to specify that name b has no instantiator (name b). We denote with $K = \{*\}$ the fact that there is no action that caused executing action.

To simplify representation of the reversible process and make manipulation with it easier, we define *history* and *general* contexts. A history context is represents the executed prefixes of a process. For instance, the process $X = \overline{b}^{*} a^{*}[i,K].\overline{c}^{*} a^{*}[i',K'].\mathbf{P}$ can be written as $X = \mathtt{H}[\mathbf{P}]$, where $\mathtt{H}[\bullet] = \overline{b}^{*} a^{*}[i,K].\overline{c}^{*} a^{*}[i',K'].\bullet$. On the top of the history context, we define a general context by adding parallel and restriction operators. For instance, process $Z \mid Y \mid X$ can be written as $C[X]$ where $C[\bullet] = Z \mid Y \mid \bullet$. Formally, we have:

Definition 39 (History and General context) *History contexts* $\mathtt{H}$ *and general contexts* C *are reversible processes with a hole* $\bullet$*, defined by the following grammar:*

$$
\mathtt{H} ::= \bullet \mid \pi[i,K].\bullet \qquad C ::= \mathtt{H}[\bullet] \ \mid\ X \mid \bullet \ \mid \nu a_{\Delta}(\bullet)
$$

Notions of bound names and bound variables in the framework are defined in usual way. There are two construct with binders: $\nu a_{\Delta}(X)$ when Δ is empty, in which the scope of name a is process X; and $b(x).\mathbf{P}$ in which scope of the variable x is process $\mathbf{P}$. We denote sets of bound names and variables of a given process X with $\mathtt{bn}(X)$ and $\mathtt{bv}(X)$, respectively. When there is no need to distinguish them, we shall write $\mathtt{bound}(X) = \mathtt{bn}(X) \cup \mathtt{bv}(X)$. Formally:

Definition 40 (Bound names and variables) *The set of the bound names and variables of the process* X*, written as* $\mathtt{bound}(X)$ *is defined by induction on*

the process structure:

$$
\begin{aligned}
&\texttt{bound}(X \mid Y) = \texttt{bound}(X) \mid \texttt{bound}(Y) \\
&\texttt{bound}(\nu a_{\Delta}(X)) = \texttt{bound}(X) && \textit{when } \Delta \textit{ is not empty} \\
&\texttt{bound}(\nu a_{\Delta}(X)) = \{a\} \cup \texttt{bound}(X) && \textit{when } \Delta \textit{ is empty} \\
&\texttt{bound}(\overline{b}a.X) = \texttt{bound}(X) \\
&\texttt{bound}(a(x).X) = \{x\} \cup \texttt{bound}(X) \\
&\texttt{bound}(\mathbf{P}) = \texttt{bound}(P) && \textit{where } P \textit{ is } \pi\textit{-calculus process}
\end{aligned}
$$

The names and variables which are not bound, are free. The set of free names and variables of the process X is denoted by $\texttt{free}(X)$.

The framework is parametric with respect to the data structure Δ which we define as an interface (in the style of a Java interface) by giving the operations that it has to offer.

Definition 41 *Δ is a data structure with the following defined operations:*

(i) $\texttt{init} : \Delta \to \Delta$ *initialises the data structure*

(ii) $\texttt{empty} : \Delta \to \texttt{bool}$ *predicate telling whether Δ is empty*

(iii) $+ : \Delta \times \mathcal{K} \to \Delta$ *operation adding a key to Δ*

(iv) $\#i : \Delta \times \mathcal{K} \to \Delta$ *operation removing a key from Δ*

(v) $\in : \Delta \times \mathcal{K} \to \texttt{bool}$ *predicate telling whether a key belongs to Δ*

In what follows, we give a brief description of a three instances of the data structure Δ: sets, sets indexed with an element and sets indexed with a set. The precise definitions of the operations on each data structure are given in Section 5.4 when we describe how these three instances will capture the three different notions of causality for π-calculus.

Set. Data structure is a set Γ containing keys (i.e. $\Gamma \subseteq \mathcal{K}$) of all the actions that extruded name a. The idea behind the memory νa_{Γ} is that **any** of the keys contained in Γ can be a contextual cause for some action having in the subject position name a. For example in the process $\nu a_{\{i_1,i_2\}}(Y \mid a(x))$ the contextual cause of the action $a(x)$ can be chosen from the set $\Gamma = \{i_1, i_2\}$.

Indexed set. Data structure is an indexed set Γ_w, where set Γ is a set containing keys ($\Gamma \subseteq \mathcal{K}$) of all the actions that extruded name a and w is the key of the first action that extruded name a. In this case contextual cause for name a is exactly w. For instance in the process $\nu a_{\{i_1,i_2\}_{i_1}}(Y \mid a(x))$ the contextual cause of the action $a(x)$ is $w = i_1$. We shall write $w = *$ if there is no action that extruded name a.

Set indexed with a set. Data structure is a set indexed with a set Γ_Ω, where Γ is a set containing keys of all the actions that extruded name a and $\Omega \in \mathcal{K}_*$ is a set containing keys of the extruders of name a which are not part of the communication. The idea behind νa_{Γ_Ω} is that the contextual cause for the name a is a set Ω. For example, in the process $\nu a_{\{i_1,i_2,i_3\}_{\{i_1,i_3\}}}(Y \mid a(x))$ the contextual cause of the action $a(x)$ is $\Omega = \{i_1, i_3\}$. We write $\Omega = \{*\}$ when there is no action that extruded name a and is not part of the synchronisation.

5.3 Operational semantics of the framework

The grammar of the labels defined on the transition $t : X \xrightarrow{\mu} Y$ is:

$$\mu ::= (i, K, j) : \alpha \qquad\qquad \alpha ::= \overline{b}a \mid b(x) \mid \overline{b}\langle \nu a_\Delta \rangle \mid \tau$$

The triple (i, K, j) contains the key i that identifies the action α, the contextual cause set $K \subseteq \mathcal{K}_*$ and the instantiator $j \in \mathcal{K}_*$ of the action α. The action α can be: standard input and output on the channel b, symbolised with $b(x)$ and $\overline{b}a$, respectively; the silent action or action $\overline{b}\langle \nu a_\Delta \rangle$ that represents the classical bound output from the π-calculus, when Δ is empty ($\texttt{empty}(\Delta) = true$), otherwise stands for free output decorated with a memory Δ.

We let $\mathcal{A}$ to be the set of all actions ranged over by α. The set of the all possible labels is defined as $\mathcal{L} = \mathcal{K} \times \mathcal{K}_* \times \mathcal{K}_* \times \mathcal{A}$. In the following definition we give the operational semantics of the framework.

Definition 42 (Operational Semantics) *The operational semantics of the reversible framework is given as a pair of two LTSs defined on the same set of reversible processes and set of labels: a forward LTS $(\mathcal{X}, \mathcal{L}, \twoheadrightarrow)$ and a backward LTS $(\mathcal{X}, \mathcal{L}, \rightsquigarrow)$. We define $\rightarrow = \twoheadrightarrow \cup \rightsquigarrow$, where $\twoheadrightarrow$ is the least transition*

relation induced by the rules in Figures 21 and 22; and $\rightsquigarrow$ *is the least transition relation induced by the rules in Figure 23.*

Now we define the function $\mathtt{key}(\cdot)$ which computes the set of keys in a given process and we specify the notion of the fresh (new) key.

Definition 43 (Process keys) *The set of* communication keys *of a process* X*, written* $\mathtt{key}(X)$*, is inductively defined as follows:*

$$\mathtt{key}(X \mid Y) = \mathtt{key}(X) \cup \mathtt{key}(Y) \qquad \mathtt{key}(\pi[i, K].X) = \{i\} \cup \mathtt{key}(X)$$
$$\mathtt{key}(\nu a_{\Delta}(X)) = \mathtt{key}(X) \qquad \mathtt{key}(\mathbf{P}) = \emptyset$$

Definition 44 *A key* i *is* fresh *in a process* X*, written* $\mathtt{fresh}(i, X)$ *if* $i \notin \mathtt{key}(X)$*.*

In the further text we give forward and the backward rules of our framework defined with a *late* semantics for inputs. Forward rules are divided into two groups, depending on whether they are common to all the instances of the framework (rules which are independent from the data structure) or they are parametric with respect to Δ.

Common rules are given in Figure 21, where $\mathtt{H}$ is a history context (Definition 39). As we can notice, in the rules OUT1 and IN1, executed actions remain in the structure of the process. Moreover, these rules generates the fresh key i and bound it to the executing action. The performed actions are annotated with the memory $[i, K]$, where i and K are the key of the action and its cause set. A prefixed process $\mathtt{H}[X]$ can perform a forward step if process X can execute it. This is depicted by the rules OUT2 and IN2. Rule for parallel composition PAR allows process to execute the action α, under the condition that key i is not used by other process in parallel ($i \notin Y$). This condition guarantees uniqueness of the action keys. Rule COM allows synchronisation between two processes in parallel which satisfy the condition $K =_* j' \ \wedge \ K' =_* j$ ($K =_* j$ stands for: $K = j$ or $* \in K$ or $j = *$; for instance, if $K = \{*, i_1\}$ and $j = i_2$, equality holds since $* \in K$). Additionally, in the rule COM, the necessary substitution is applied to the continuation of the input process in the following way: every occurrence of variable $x \in \mathtt{fn}(Y')$ is substituted with the name a decorated with the key i of the executed action. In this way,

$$(\textsc{Out1})\ \overline{b}^j a^{j_1}.\mathbf{P} \xrightarrow{(i,K,j):\overline{b}a} \overline{b}^j a^{j_1}[i,K].\mathbf{P}$$

$$(\textsc{Out2})\ \frac{X \xrightarrow{(i,K,j):\overline{b}a} X' \quad \mathtt{fresh}(i,\mathtt{H}[X])}{\mathtt{H}[X] \xrightarrow{(i,K,j):\overline{b}a} \mathtt{H}[X']}$$

$$(\textsc{In1})\ b^j(x).\mathbf{P} \xrightarrow{(i,K,j):b(x)} b^j(x)[i,K].\mathbf{P}$$

$$(\textsc{In2})\ \frac{X \xrightarrow{(i,K,j):b(x)} X' \quad \mathtt{fresh}(i,\mathtt{H}[X])}{\mathtt{H}[X] \xrightarrow{(i,K,j):b(x)} \mathtt{H}[X']}$$

$$(\textsc{Par})\ \frac{X \xrightarrow{(i,K,j):\alpha} X' \quad i \notin Y \quad \mathtt{bound}(\alpha) \cap \mathtt{free}(Y) = \emptyset}{X \mid Y \xrightarrow{(i,K,j):\alpha} X' \mid Y}$$

$$(\textsc{Com})\ \frac{X \xrightarrow{(i,K,j):\overline{b}a} X' \quad Y \xrightarrow{(i,K',j'):b(x)} Y' \quad K =_* j' \ \wedge\ K' =_* j}{X \mid Y \xrightarrow{(i,*,*):\tau} X' \mid Y'\{^{a^i}/_x\}}$$

Figure 21: Common rules for all instances of the framework.

future computations of the process $Y'\{^{a^i}/_x\}$ will be aware that variable x was substituted with the name a during the synchronisation identified by i. In a^i, the key i is called the instantiator and used only to track the substitution, not to define a name. For instance, having the reversible process $\overline{b}^j a^*.\mathbf{P} \mid b^{j'}(x).\mathbf{P}'$, the communication between them is allowed even if they do not have the same instantiators on the channel b.

In order to have better intuition about the rules presented above, we give the following example.

Example 21 Let $X = \overline{b}^* a^*.\mathbf{0} \mid b^*(x).\overline{x}c^*$ be a reversible process. Process X has two possibilities for executing forward actions:

- an output action $\overline{b}a$ and an input action $b(x)$ can be performed. In

$$(\textsc{Cause Ref})\ \frac{X \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow X' \quad a \in sub(\alpha) \qquad \texttt{empty}(\Delta) \neq true \quad \texttt{Cause}(\Delta, K, K')}{\nu a_\Delta(X) \xrightarrow{(i,K',j):\alpha}\!\!\!\!\rightarrow \nu a_\Delta(X'_{[K'/K]@i})}$$

$$(\textsc{Open})\ \frac{X \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow X' \quad \alpha = \overline{b}a \vee \alpha = \overline{b}\langle \nu a_{\Delta'}\rangle \quad \texttt{Update}(\Delta, K, K')}{\nu a_\Delta(X) \xrightarrow{(i,K',j):\overline{b}\langle \nu a_\Delta\rangle}\!\!\!\!\rightarrow \nu a_{\Delta+i}(X'_{[K'/K]@i})}$$

$$(\textsc{Close})\ \frac{X \xrightarrow{(i,K,j):\overline{b}\langle \nu a_\Delta\rangle}\!\!\!\!\rightarrow X' \quad Y \xrightarrow{(i,K',j'):b(x)}\!\!\!\!\rightarrow Y' \quad K =_* j' \ \wedge\ K' =_* j}{X \mid Y \xrightarrow{(i,*,*):\tau}\!\!\!\!\rightarrow \nu a_\Delta(X'_{\#i} \mid Y'\{^{a^i}/_x\})}$$

$$(\textsc{Res})\ \frac{X \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow X' \quad a \notin \alpha}{\nu a_\Delta(X) \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow \nu a_\Delta(X')}$$

Figure 22: Parametric rules

this case actions synchronise with the environment:

$$\overline{b}^* a^*.\mathbf{0} \mid b^*(x).\overline{x}c^* \xrightarrow{(i,*,*):\overline{b}a}\!\!\!\!\rightarrow \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x).\overline{x}c^*$$
$$\xrightarrow{(i',*,*):b(x)}\!\!\!\!\rightarrow \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i',*].\overline{x}c^* = Y_1$$

As we can notice, the output action $\overline{b}a$ is identified by key i, while the input action is identified by key i'.

- Two parallel components of the process X can synchronise over the channel b, and we have:

$$\overline{b}^* a^*.\mathbf{0} \mid b^*(x).\overline{x}c^* \xrightarrow{(i,*,*):\tau}\!\!\!\!\rightarrow \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i,*].\overline{a}^i c^* = Y_2$$

  We can notice that during the synchronisation identified with key i, variable x was substituted with the received name a decorated with the key i. In this way substitution of a name is recorded.

Parametric rules are represented in Figure 22. Depending on the used data structure, the mechanism for choosing a contextual cause differs. For this reason, we introduce two new predicates $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$. We will give the precise definitions for the predicates when we discuss how

to map different causal semantics into the framework (Section 5.4). The intuition behind the predicates is that they define how contextual cause is chosen from the memory delta. Hence, by instantiating Δ with the specific data structure, $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$ need to be implemented differently.

Every time when action α, with $a \in \alpha$, is passing the restriction νa_{Δ}, it is necessary to check if the contextual cause set needs to be modified. If name a is in the subject position of the label α and $\texttt{empty}(\Delta) = false$, then rule CAUSE REF is used, otherwise, if name a is in the object position, rule OPEN is applied. Rule CAUSE REF can be used only if name a was already extruded by some other action in the past and in this case predicate $\texttt{Cause}(\Delta, K, K')$ ensures that contextual cause set K will be substituted with new cause set K'. Additionally predicate $\texttt{Cause}(\Delta, K, K')$ gives definition of the cause set K'. The *contextual cause update* operation defined on the process X, written as $X_{[K'/K]@i}$ updates the contextual cause K of the action identified by i with the new cause K'. Formally:

Definition 45 (Contextual Cause Update) *The* contextual cause update *of the process X, written $X_{[K'/K]@i}$ is defined as follows:*

$$(X \mid Y)_{[K'/K]@i} = X_{[K'/K]@i} \mid Y_{[K'/K]@i} \qquad \texttt{H}[\pi[i, K].X]_{[K'/K]@i} = \texttt{H}[\pi[i, K'].X]$$
$$(\nu a_{\Delta}(X))_{[K'/K]@i} = \nu a_{\Delta}(X)_{[K'/K]@i} \qquad \texttt{H}[\pi[j, K].X]_{[K'/K]@i} = \texttt{H}[\pi[j, K].X]$$

We use this operation in rules CAUSE REF and OPEN. Restricted name can be sent out to the environment (extruded) by applying the rule OPEN. In this case, we need to record what was the key of the action that extruded restricted name. For this reason key i is added into the memory Δ. The predicate $\texttt{Update}(\Delta, K, K')$ in the rule OPEN defines what will be the new contextual cause set K'. Rule CLOSE allows synchronisation between two processes when bound output is included. Additional condition on the rule, needs to be satisfied. In the resulting process, after execution of the τ-action, we can notice the operator $_{\#i}$ defined on the data structure. Different data structures requires different implementations of it (Section 5.4). Here we give just intuition: operator $_{\#i}$ removes from data structure Δ, the keys of the actions that extruded restricted name but are part of synchronisations. This is necessary, since in causal semantics [14; 19],

τ-actions do not bring the causal information. Rule RES is defined in the usual way. Better intuition of how parametric rules work, will be given by means of examples in Section 5.4.

Backward rules are presented in Figure 23 and they are symmetric to the forward ones. To simplify the proofs, we keep the predicates, even if they are not necessary for the backward transition (there is no non-determinism while reversing the actions, since every action is identified by unique key).

In order to have better understanding of the backward rules, we shall give the following example.

Example 22 We consider the following processes from Example 21:

- $Y_1 = \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i',*].\overline{x}c^*$; The backward actions that can be performed by the process Y_1 are: the input on the channel b identified with key i' and the output on the channel b identified with key i. These backward steps can be executed in any order, for example, we undo first the input and then the output:

$$Y_1 = \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i',*].\overline{x}c^* \xrightarrow{(i',*,*):b(x)} \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x).\overline{x}c^*$$
$$\xrightarrow{(i,*,*):\overline{b}a} \overline{b}^* a^*.\mathbf{0} \mid b^*(x).\overline{x}c^* = X$$

  The history part of the process Y_1 has all necessary information to reverse those two actions.

- $Y_2 = \overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i,*].\overline{a}^i c^*$; The only possible backward step for process Y_2 is undoing the synchronisation done over the channel b:

$$\overline{b}^* a^*[i,*].\mathbf{0} \mid b^*(x)[i,*].\overline{a}^i c^* \xrightarrow{(i,*,*):\tau} \overline{b}^* a^*.\mathbf{0} \mid b^*(x).\overline{x}c^* = X$$

  As we can notice, the substitution is also reversed and name a^i is substituted with variable x as it was in the initial state of the process X.

Remark 7 The framework can be easily equipped with the choice operator $(+)$ by making the operator static as in [70].

$$(\textsc{Out1}^\bullet)\ \overline{b}^j a^{j_1}[i,K].\mathbf{P} \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} \overline{b}^j a a^{j_1}.\mathbf{P}$$

$$(\textsc{Out2}^\bullet)\ \frac{X' \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} X \quad \texttt{fresh}(i, \texttt{H}[X])}{\texttt{H}[X'] \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} \texttt{H}[X]}$$

$$(\textsc{In1}^\bullet)\ b^j(x)[i,K].\mathbf{P} \overset{(i,K,j):b(x)}{\rightsquigarrow} b^j(x).\mathbf{P}$$

$$(\textsc{In2}^\bullet)\ \frac{X' \overset{(i,K,j):b(x)}{\rightsquigarrow} X \quad \texttt{fresh}(i, \texttt{H}[X])}{\texttt{H}[X'] \overset{(i,K,j):b(x)}{\rightsquigarrow} \texttt{H}[X]} \qquad (\textsc{Par}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad i \notin Y}{X' \mid Y \overset{(i,K,j):\alpha}{\rightsquigarrow} X \mid Y}$$

$$(\textsc{Res}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad a \notin \alpha}{\nu a_\Delta(X') \overset{(i,K,j):\alpha}{\rightsquigarrow} \nu a_\Delta(X)}$$

$$(\textsc{Com}^\bullet)\ \frac{X' \overset{(i,K,j):\overline{b}a}{\rightsquigarrow} X \quad Y' \overset{(i,K',j'):b(x)}{\rightsquigarrow} Y \quad K =_* j' \ \wedge\ K' =_* j}{X' \mid Y' \overset{(i,*,*):\tau}{\rightsquigarrow} X \mid Y\{^x/_{a^i}\}}$$

$$(\textsc{Open}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad \alpha = \overline{b}a \vee \alpha = \overline{b}\langle \nu a_{\Delta'}\rangle \quad \texttt{Update}(\Delta, K, K')}{\nu a_{\Delta+i}(X') \overset{(i,K',j):\overline{b}\langle \nu a_\Delta\rangle}{\rightsquigarrow} \nu a_\Delta(X)}$$

$$(\textsc{Cause Ref}^\bullet)\ \frac{X' \overset{(i,K,j):\alpha}{\rightsquigarrow} X \quad a \in sub(\alpha) \qquad \texttt{empty}(\Delta) \neq true \quad \texttt{Cause}(\Delta, K, K')}{\nu a_\Delta(X') \overset{(i,K',j):\alpha}{\rightsquigarrow} \nu a_\Delta(X)}$$

$$(\textsc{Close}^\bullet)\ \frac{X' \overset{(i,K,j):\overline{b}\langle \nu a_\Delta\rangle}{\rightsquigarrow} X \quad Y' \overset{(i,K',j'):b(x)}{\rightsquigarrow} Y \quad K =_* j' \ \wedge\ K' =_* j}{\nu a_\Delta(X' \mid Y') \overset{(i,*,*):\tau}{\rightsquigarrow} X \mid Y\{^x/_{a^i}\}}$$

Figure 23: Backward rules.

5.4 Mapping three causal semantics

In this Section we show how causality notions induced by three different causal semantics [14; 19; 20] can be mapped in our framework. We do it by defining the operators from Definition 41 for the specific data structure, and by giving definitions for predicates in Figure 22.

5.4.1 Reversible semantics for the π-calculus

A compositional semantics for the reversible π-calculus, given by Cristescu et al [20] is revised in Section 4.3. History information is kept in the memory attached to every process. A reversible process is of the form $m \triangleright P$, where m is a memory and P is the standard π-calculus process. The extruder information is kept in the construct νa_Γ, which behaves as a classical π-calculus restriction when set Γ is empty, otherwise it serves as a memory.

Naturally, to capture this behaviour, we shall use *set* Γ as data structure Δ. We now implement the operations of Definition 41 as follows:

Definition 46 (Operations on a set) *The operations on a set Γ are defined as:*

(i) $\texttt{init}(\Gamma) = \emptyset$

(ii) $\texttt{empty}(\Gamma) = true$, *when* $\Gamma = \emptyset$

(iii) $+$ *is the classical addition of elements to a set*

(iv) $\#i$ *is defined as the identity, that is* $\Delta_{\#i} = \Gamma_{\#i} = \Gamma$.

(v) $i \in \Gamma$ *the key* i *belongs to the set* Γ

Data structure is initialised when $\Gamma = \emptyset$ and it implies that $\texttt{empty}(\Gamma) = true$, as expected. The operation $_{\#i}$ is define as identity, while $+$ and $i \in \Gamma$ are classical operations defined on the set.

To capture notion of causality introduced in [20], we need to adapt definition of the instantiation relation (Definition 37) to our framework and define it on the past prefixes. For instance in the process

$$b^*(x)[i_1, K_1].\overline{a}^{i_1} c^*[i_2, K_2].Y$$

actions i_1 and i_2 are in instantiation relation, since action i_1 instantiate name a. We can see it in the past prefix $\overline{a}^{i_1}c^*[i_2, K_2]$ where name a is decorated with the key i_1. Formally, the instantiation relation on the prefixes is defined as follows.

Definition 47 (Instantiation relation on the framework) *Two keys i_1 and i_2 such that $i_1, i_2 \in \mathtt{key}(X)$ and $X = C[b^{j_1}(x)[i_1, K_1].Y]$ with $Y = C'[\pi[i_2, K_2].Z]$ where $j_2 \in \pi$, are in* instantiation relation, *denoted with $i_1 \rightsquigarrow_X i_2$, if $j_2 = i_1$. If $i_1 \rightsquigarrow_X i_2$ holds, we will write $K_1 \rightsquigarrow_X K_2$.*

Note that since the actions in [20] can be caused only through the subject of the label, contextual cause set K is a singleton. Now we can give the last definition necessary to obtain the $R\pi$ causality, definitions of the predicates from the rules in Figure 22.

Definition 48 ($R\pi$ causality) *To capture $R\pi$ causality, data structure is instantiated with the set Γ and the predicates from Figure 22 are defined as:*

1. $\mathtt{Cause}(\Gamma, K, K')$ *stands for $K' = K$ or $\exists K' \in \Gamma\ K \rightsquigarrow_X K'$;*

2. $\mathtt{Update}(\Gamma, K, K')$ *stands for $K' = K$.*

The predicate $\mathtt{Update}(\Gamma, K, K')$, used in rule CAUSE REF means that the new cause K' is the same as the old one, denoted with K. The first predicate defined above, $\mathtt{Cause}(\Gamma, K, K')$, is used in the rule CAUSE REF and it defines how the new contextual cause K' is chosen. To illustrate this, we give the following examples.

Example 23 Let us consider the process $X = \nu a_\emptyset(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$ where we have parallel extrusion of the same name a. By extruding the name a with the rule OPEN twice, on the channels b and c, we obtain a process:

$$\nu a_{\{i_1,i_2\}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^*[i_2, *] \mid a^*(x))$$

The rule CAUSE REF is used for the execution of the third action where action $a(x)$ can choose its cause from the set $\{i_1, i_2\}$. By choosing, for example, i_2 as a cause, we obtain the process:

$$\nu a_{\{i_1,i_2\}}(\overline{b}^* a[i_1, *] \mid \overline{c}^* a[i_2, *] \mid a^*(x)[i_3, i_2])$$

In the memory $[i_3, i_2]$ we can see that the action identified with key i_3 needs to be reversed before the action with key i_2. Process $\overline{b}^* a[i_1, *]$ can execute a backward step at any time with the rule OPEN$^\bullet$.

Let us show one more example where instantiation relation is used while choosing the new cause K'.

Example 24 Consider the process $X = \nu a_{\{i_3,4\}}(\nu a_{\{i_1,i_2\}}(a^*(x)))$ where $i_1 \rightsquigarrow_X i_3$ and $i_2 \rightsquigarrow_X i_4$. For the sake of simplicity, we omitted the history part of the process X. By passing the restriction operator $\nu a_{\{i_1,i_2\}}$, action $a(x)$ can choose between two causes, i_1 and i_2 (by applying the rule CAUSE REF). Let assume that i_1 is chosen. To pass the restriction $\nu a_{\{i_3,i_4\}}$, one more time rule CAUSE REF is applied and we have:

$$\frac{\nu a_{\{i_1,i_2\}}(a^*(x)) \xrightarrow{(i,i_1,*):a(x)}\!\!\!\!\rightarrow \nu a_{\{i_1,i_2\}}(a^*(x)[i,i_1]) \quad i_3 \in \Gamma\ i_1 \rightsquigarrow_X i_3}{\nu a_{\{i_3,i_4\}}(\nu a_{\{i_1,i_2\}}(a^*(x))) \xrightarrow{(i,i_3,*):a(x)}\!\!\!\!\rightarrow \nu a_{\{i_3,i_4\}}(\nu a_{\{i_1,i_2\}}(a^*(x)[i,i_3]))}$$

where we can notice that in the premises, contextual cause was i_1 and that is saved in the resulting process $\nu a_{\{i_1,i_2\}}(a^*(x)[i,i_1])$. Passing the restriction $\nu a_{\{i_3,i_4\}}$, action $a(x)$ needs to take another cause, since $i_1 \notin \{i_3, i_4\}$ and the chosen cause is $K' = i_3$ since $i_1 \rightsquigarrow_X i_3$. Contextual cause i_3 is then recorded in the final process $\nu a_{\{i_3,i_4\}}(\nu a_{\{i_1,i_2\}}(a^*(x)[i,i_3]))$.

5.4.2 Boreale and Sangiorgi causal semantics

A compositional causal semantics for standard π-calculus was introduced by Boreale and Sangiorgi [14]. We have revised this causal semantics in Section 4.1, where we did not take into account replication operator and we define it with late (rather than early, as it was originally given) semantics. In this section we describe how to capture the notion of causality induced by [14] with our framework.

The authors separated dependences between the actions in two groups: *subject* and the *object* dependency. The subject one is captured by the causal term $\mathtt{K} :: A$, where the set of causes $\mathtt{K}$ records that every action performed by A depends on $\mathtt{K}$. The object dependency is given on the trace of the process (Definition 28) and intuitively it says that: the first input action with the variable x, let say $a(x)$, shall cause every other action using variable x in any position of the label (this part of definition is captured

in the framework by structure of the process. A more detailed discussion about it can be find in Section 5.6); the first action that extrudes restricted name shall cause every future action using that name in any position of the label (subject or the object). Since we have that only the first extrusion of bound name will cause the the rest of the actions using that name, we shall use *indexed set* Γ_w for the data structure Δ. The operations given in Definition 41 that data structures offers, are defined on the indexed set in the following way.

Definition 49 (Operations on an indexed set) *The operations on an indexed set Γ_w are defined as:*

(i) $\texttt{init}(\Gamma_w) = \emptyset_*$

(ii) $\texttt{empty}(\Gamma_w) = true$, *when* $\Gamma = \emptyset \ \wedge \ w = *$

(iii) *operation* $+$ *is defined as:* $\Gamma_w + i = \begin{cases} (\Gamma \cup \{i\})_i, \text{ when } w = * \\ (\Gamma \cup \{i\})_w, \text{ when } w \neq * \end{cases}$

(iv) *operation* $\#i$ *is defined inductively as:*

$$(X \mid Y)_{\#i} = X_{\#i} \mid Y_{\#i} \quad (\texttt{H}[X])_{\#i} = \texttt{H}[X_{\#i}] \qquad (\mathbf{P})_{\#i} = \mathbf{P}$$
$$(\nu a_{\Gamma_i} X)_{\#i} = \nu a_{\Gamma_*} X_{\#i} \quad (\nu a_{\Gamma_w} X)_{\#i} = \nu a_{\Gamma_w} X_{\#i}$$

(v) $i \in \Gamma_w$ *the key* i *belongs to the set* Γ, *regardless of* w *(e.g.* $i \in \{i\}_*$*)*

The data structure is initialised when $\texttt{init}(\Gamma_w) = \emptyset_*$ and it implies that $\texttt{empty}(\Gamma_w) = true$, as expected. The $+$ operator is defined as: the key added to the Γ_w will be added to the set Γ and on the place of w if $w = *$, otherwise it will be added just to the set Γ. For instance, after adding key i_3 to the $\{i_1, i_2\}_*$ we obtain $\{i_1, i_2, i_3\}_{i_3}$. Operation $_{\#i}$, substitute value of w in Γ_w with element $*$, when $w = i$. For example, the result of applying operation $_{\#i}$ on $\{i, i_1\}_i$ is $\{i, i_1\}_*$. The operation of belonging is defined on the set Γ in the classical way, regardless the index w. For instance, the key i belongs to the indexed set $\{i_1, i\}_{i_1}$.

To capture the causality defined in Section 4.1, we give definitions for the predicates in Figure 22.

Definition 50 (Boreale and Sangiorgi causal semantics) *To capture Borale and Sangiorgi causality the data structure Δ is instantiated with indexed set Γ_w and the predicates from Figure 22 are defined as:*

1. $\texttt{Cause}(\Gamma_w, K, K')$ *stands for* $K' = K \cup \{w\}$

2. $\texttt{Update}(\Gamma_w, K, K')$ *stands for* $K' = K \cup \{w\}$

From the object causality definition, we have that an output action can be caused through the subject and object position of a label. For instance, consider a process $\nu a(\nu b(\overline{c}b \mid \overline{d}a \mid \overline{b}a))$ and its trace $\xrightarrow{\overline{c}\langle \nu b\rangle}\xrightarrow{\overline{d}\langle \nu a\rangle}\xrightarrow{\overline{b}a}$. The action $\overline{b}a$ depends on the both actions executed before (on the first one because it extrudes the name b and on the second one because it extrudes name a). For this reason, predicates $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$ define the new cause set K' by adding the value of w to the old set of the causes K. In this way, both causes from the example before will be saved in K'.

We remark that silent actions do not exhibit or impose contextual causes. We give the following example to have a better intuition about framework expressing Boreale and Sangiorgi causality.

Example 25 Consider the process $X = \nu a_{\emptyset_*}(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$. By extruding the name a over the channel b (rule OPEN is applied), we obtain the process:

$$\nu a_{\{i_1\}_{i_1}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^* \mid a^*(x))$$

From the memory $\{i_1\}_{i_1}$ we have that $w = i_1$. By executing the actions $\overline{c}a$ with the rule OPEN and $a(x)$ with the rule CAUSE REF, predicates on the rules $\texttt{Update}(\cdot)$ and $\texttt{Cause}(\cdot)$ ensure that key $i_1 = w$ will be added to cause sets. We obtain the process:

$$\nu a_{\{i_1,i_2\}_{i_1}}(\overline{b}^* a^*[i_1, *] \mid \overline{c}^* a^*[i_2, \{i_1, *\}] \mid a^*(x)[i_3, \{i_1, *\}])$$

In the memories $[i_2, \{i_1, *\}]$ and $[i_3, \{i_1, *\}]$ we can notice that the executed actions are caused by action i_1 and for this reason the action i_1 needs to be reversed last. The actions i_2 and i_3 can be reversed in any order.

5.4.3 Crafa, Varacca and Yoshida causal semantics

A compositional event structure semantics for the forward π-calculus is introduced in [19]. In Section 4.2 we revised the causal semantics given in [19], where we use as the base the π-calculus defined in Figure 12. The process is represented as a pair $(\mathbf{E}, \mathbf{X})$, where $\mathbf{E}$ is the prime event structure and $\mathbf{X}$ is a set of bound names. The object causality on events (Definition 32) is defined as: the action with a restricted name in the subject position, can be executed if at least one of the actions that extruded restricted name was executed in the past. It is not necessary to record which one. In case of parallel extrusion of the same name, for instance in process $\nu a(\overline{b}a \mid \overline{c}a \mid a(x))$ action $a(x)$ can be caused by any of the extrusions (configurations $\{\overline{b}a, a(x)\}$ and $\{\overline{c}a, a(x)\}$ are both permitted), without recording the actual extruder.

The consequence of this approach is that events do not have unique causal history. In [22] authors discussed that this type of causality cannot be expressed when processes with contexts are considered. For example, in the process $P = \nu a(\overline{b}a \mid \overline{c}a \mid a(x))$, the cause of the action $a(x)$ is either $\overline{b}a$ or $\overline{c}a$ but if context is added to this process (i.e. we consider close term), the ambiguity of the cause choice is lost. The choice of the cause is determined by the context (for instance, we can add context $b(y).\overline{y}d$ to the process P and in this case we know that action $a(x)$ is caused by $\overline{b}a$).

In our framework, reversibility is causally-consistent, hence we need to keep track of the causes. If not, by executing backward actions, we could reach a state that is not consistent (not consistent state would be the state where the action that extrude restricted name is reversed, while action that using that name in the subject position is not). For this reason, we consider two possibilities for keeping track of causes: the first option is to choose one of the possible extruders and save it in the history of the process and the second one would be to record all of them that happened before the action with a restricted name in the subject position. If we would go with the first approach, we would obtain a notion of causality that is similar to causality given in [20]. In the following we concentrate on the second approach with the idea that since we do not know the exact

extruder that caused the action with bound subject, we record the whole set of them that happened in the past. Hence, the data structure that can be used to represent the whole set of extruders is: *set indexed with a set* Γ_Ω. The reason why two sets are necessary is because the τ-actions do not impose causes. Every extruder will be recorded in the set Γ and in that way we will keep track about the scope of the restricted name, while extruders which are not part of synchronisations will be saved in Ω. The operations given in Definition 41 that data structures offers, are defined on the set indexed with a set in the following way.

Definition 51 (Operations on a set indexed with a set) *The operations on a set indexed with a set (Γ_Ω) are defined as:*

(i) $\texttt{init}(\Gamma_\Omega) = \emptyset_{\{*\}}$

(ii) $\texttt{empty}(\Gamma_\Omega) = true$, *when* $\Gamma = \emptyset \ \wedge \ \Omega = \{*\}$

(iii) *operation* $+$ *is defined as:* $(\Gamma_\Omega) + i = (\Gamma \cup \{i\})_{(\Omega \cup \{i\})}$

(iv) *operation* $_{\#i}$ *is defined inductively as:*

$$(X \mid Y)_{\#i} = X_{\#i} \mid Y_{\#i} \qquad (\nu a_{\Gamma_\Omega} X)_{\#i} = \nu a_{\Gamma_{\Omega \setminus \{i\}}} X_{\#i}$$
$$(\texttt{H}[X])_{\#i} = \texttt{H}[X_{\#i}] \qquad (\mathbf{P})_{\#i} = \mathbf{P}$$

(v) $i \in \Gamma_\Omega$ *the key* i *belongs to the set* Γ, *regardless* Ω *(e.g.* $i \in \{i\}_{\{*\}}$*)*

The data structure is initialised when $\texttt{init}(\Gamma_\Omega) = \emptyset_{\{*\}}$ and it implies that $\texttt{empty}(\Gamma_{\{*\}}) = true$. The $+$ operator is defined by adding the key into both sets. For instance, the result of adding key i to the data structure $\{i_1, i_2\}_{\{i_2\}}$ is $\{i_1, i_2, i\}_{\{i_2, i\}}$. The operation $_{\#i}$, removes the key i from the set Ω. For example, if we apply operation $_{\#i}$ to the data structure $\{i_1, i_2, i\}_{\{i_2, i\}}$, we obtain $\{i_1, i_2, i\}_{\{i_2\}}$ (the key i is deleted from the set $\Omega = \{i_2, i\}$). The operation of belonging is defined on the set Γ in classical way, regardless to the set Ω. For instance, the key i belongs to the data structure $\{i_1, i\}_{\{i_1\}}$.

To obtain revised causality introduced in [19] where we keep track of the extruders in the way described above, we need to give definition of the predicates in Figure 22.

Definition 52 (Revised Crafa, Varacca and Yoshida causal semntics) *If an indexed set Γ_Ω is chosen as a data structure for a memory Δ, the predicates are defined as:*

1. $\texttt{Cause}(\Gamma_\Omega, K, K')$ *stands for* $K' = K \cup \Omega$
2. $\texttt{Update}(\Gamma_\Omega, K, K')$ *stands for* $K' = K$

In the definition above we can notice that the new cause K' in the predicate $\texttt{Cause}(\Gamma_\Omega, K, K')$, gathers all extrusions of a restricted name executed previously, which are not part of the synchronisation. With the following example we give the idea of how the mechanism works.

Example 26 Let us consider the process $X = \nu a_{\emptyset_{\{*\}}}(\overline{b}^* a^* \mid \overline{c}^* a^* \mid a^*(x))$. By extruding name a on the channels b and c (by applying rule OPEN), we obtain the process:

$$\nu a_{\{i_1,i_2\}_{\{*,i_1,i_2\}}}(\overline{b}^* a^*[i_1,*] \mid \overline{c}^* a^*[i_2,*] \mid a^*(x))$$

As we can notice, keys i_1 and i_2 are added in the data structure Γ_Ω in both sets. From Definition 52 we have that the cause of the action $a(x)$ will be the whole set $\{*, i_1, i_2\}$. By executing the input action we obtain process:

$$\nu a_{\{i_1,i_2\}_{\{*,i_1,i_2\}}}(\overline{b}^* a[i_1,*] \mid \overline{c}^* a[i_2,*] \mid a^*(x)[i_3,\{*,i_1,i_2\}])$$

From reversible point of view, action $a(x)$ needs to be reversed first one (we can notice it in the memory $[i_3, \{*, i_1, i_2\}]$). The other two actions can be reversed in any order.

5.5 Properties of the framework

In this Section we show that our framework enjoys typical properties of reversible process calculi [20; 23; 51; 70]: correspondence with the base calculus (in our case it is π-calculus and it is given in Section 5.5.1); property declaring that every step can be undone, what is in the literature usually stated as Loop Lemma (Lemma 18); permutation of the concurrent transitions, specified as Square (Diamond) Lemma (Lemma 20) and property stating that reversibility induced by the framework is causally consistent (Theorem 10). All the properties defined on the framework are limited to the reachable processes given with the following definition.

Definition 53 (Initial and Reachable process) *A reversible process X is* initial *if it is derived from a π-calculus process P where all the restricting operators are initialised and in every prefix, names are decorated with a distinguished symbol $*$. A reversible process is* reachable *if it can be derived from an initial process by using the rules in Figures 21, 22 and 23.*

5.5.1 π-calculus correspondence

We start by showing that our framework is conservative extension of the π-calculus. First we define an erasing function φ which removes past information from a given reversible process X and in that way obtains a π-calculus process. After that we show a *forward* operational correspondence between a reversible process X and its projection on the π-calculus $\varphi(X)$.

Definition 54 (Erasing function) *Let $\mathcal{P}$ and $\mathcal{X}$ be the sets of π-calculus and reversible processes, respectively. The function $\varphi : \mathcal{X} \to \mathcal{P}$ that maps reversible processes to the π-calculus, is inductively defined as follows:*

$$\begin{array}{lll}
\varphi(X \mid Y) = \varphi(X) \mid \varphi(Y) & & \varphi(\overline{b}^j a^{j'}.\mathbf{P}) = \overline{b}a.\varphi(\mathbf{P}) \\
\varphi(\nu a_\Delta(X)) = \varphi(X) & \textit{if } \texttt{empty}(\Delta) = false & \varphi(b^j(x).\mathbf{P}) = b(x).\varphi(\mathbf{P}) \\
\varphi(\nu a_\Delta(X)) = \nu a\ \varphi(X) & \textit{if } \texttt{empty}(\Delta) = true & \varphi(\mathbf{0}) = \mathbf{0} \\
\varphi(\texttt{H}[X]) = \varphi(X) & &
\end{array}$$

The erasing function can be extended to labels as:

$$\begin{array}{lll}
\varphi((i, K, j) : \alpha) = \varphi(\alpha) & & \varphi(\overline{b}a) = \overline{b}a \\
\varphi(\overline{b}\langle \nu a_\Delta \rangle) = \overline{b}\langle \nu a \rangle & \textit{when } \texttt{empty}(\Delta) = true & \varphi(b(x)) = b(x) \\
\varphi(\overline{b}\langle \nu a_\Delta \rangle) = \overline{b}a & \textit{when } \texttt{empty}(\Delta) = false & \varphi(\tau) = \tau
\end{array}$$

As we can notice in the definition above, erasing function eliminates history of the process (past prefixes) and restriction operators νa_Δ when Δ is not empty. Additionally, it removes instantiators from the process that needs to be executed ($\mathbf{P}$). From the labels, function φ deletes the triple (i, K, j) and transform bound $\overline{b}\langle \nu a_\Delta \rangle$ to the free output, when Δ is not empty.

In what follows, we give the forward operational correspondence between reversible process X and π-calculus process $\varphi(X)$. We start by

showing that every forward action that reversible process perform can be mimic by the π-calculus. We shall use $\twoheadrightarrow_\pi$ to specify that transition belong to the semantics of the π-calculus.

Lemma 16 *If exists a transition $X \xrightarrow{\mu}\!\!\!\!\rightarrow Y$ then $\varphi(X) \xrightarrow{\varphi(\mu)}\!\!\!\!\rightarrow_\pi \varphi(Y)$.*

Proof The proof is by induction on the derivation tree of the transition $X \xrightarrow{\mu}\!\!\!\!\rightarrow Y$ with the case analysis on the last applied rule. □

Now we give the converse of Lemma 16.

Lemma 17 *If there is a transition $P \xrightarrow{\varphi(\mu)}\!\!\!\!\rightarrow_\pi Q$ then for all reachable X such that $\varphi(X) = P$, there is a transition $X \xrightarrow{\mu}\!\!\!\!\rightarrow Y$ with $\varphi(Y) = Q$.*

Proof The proof is by induction on the derivation tree of the transition $P \xrightarrow{\varphi(\mu)}\!\!\!\!\rightarrow_\pi Q$. □

By combining the two previous lemmata we can state that the relation between a process X and its corresponding π term P is a strong bisimulation. Formally:

Corollary 1 *The relation given by $(X, \varphi(X))$, for all reachable processes X, is a strong bisimulation.*

5.5.2 Causal-consistency of the reversible framework

In this Section we show that reversibility induced by our parametric framework is defined correctly. Most of the proof outlines are adapted from [20; 23] with more complex disscusions due to the generality of our framework. The first important property is stating that every transition (reduction step) can be undone. In the literature it is depicted with the so-called Loop Lemma ([23, Lemma 6]) which we adapt to the framework as:

Lemma 18 (Loop Lemma) *For every reachable process X and forward transition $t : X \xrightarrow{\mu}\!\!\!\!\rightarrow Y$ there exists a backward transition $t' : Y \overset{\mu}{\rightsquigarrow} X$, and conversely.*

Proof The proof follows from the symmetry of the forward and the backward rules. □

Followed by the symmetry of the rules, we define reverse transitions.

Definition 55 (Reverse transition) *The reverse transition of a transition* $t : X \overset{\mu}{\twoheadrightarrow} Y$*, written* $t^\bullet$*, is the transition with the same label and the opposite direction* $t^\bullet : Y \overset{\mu}{\rightsquigarrow} X$*, and vice versa. Thus* $(t^\bullet)^\bullet = t$*.*

An important component for reversibility is *causal relation* between the actions. It is possible to reverse an action α only if all the actions that were caused by α, were reversed previously. In this way, one avoids to reach states that are not consistent. For instance, in the reversible process $\nu a_{\{i_1\}}(\overline{b}a[i_1, *] \mid a(x)[i_2, \{i_1\}])$, obtained from the π-calculus process $\nu a(\overline{b}a \mid a(x))$, if the action with a key i_1 would be reversed first, we reach the state that is not consistent (it is impossible to have a state in which action with a bound subject is executed before the restricted name was extruded and became a free name).

In the following, we give a definition of the causality relation on the framework, regardless of the used data structure. It is represented as the union of the *structural* and *object* causality. We start by defining structural dependences between two past prefixes. For instance, in the reversible process $X = \overline{b}a[i, K].\overline{c}d[i', K']$, past prefix with key i is structural cause of the past prefix with key i'. Formally:

Definition 56 (Structural cause on the past prefixes) *For every two keys* i_1 *and* i_2 *such that* $i_1, i_2 \in \texttt{key}(X)$*, we say that past prefix with the key* i_1 *is a* structural cause *of the past prefix with the key* i_2 *in the process* X*, written as* $i_1 \sqsubset_X i_2$ *if* $X = C[\pi[i_1, K_1].Y]$ *and* $i_2 \in \texttt{key}(Y)$*.*

We extend definition of the structural cause on the past prefixes to the transitions.

Definition 57 (Structural causality) *Transition* $t_1 : X \xrightarrow{(i_1, K_1, j_1):\alpha_1} X'$ *is a* structural cause *of transition* $t_2 : X'' \xrightarrow{(i_2, K_2, j_2):\alpha_2} X'''$*, written* $t_1 \sqsubset t_2$*, if* $i_1 \sqsubset_{X'''} i_2$ *or* $i_2 \sqsubset_X i_1$ *if transitions are backward.* Structural causality*, denoted with* $\sqsubseteq$*, is obtained by reflexive and transitive closure of* $\sqsubset$*.*

Now we show some examples of structural causality between transitions.

Example 27 Let us consider the process $X = \overline{b}a.\overline{c}d$ and forward transitions:

$$t_1 : X \xrightarrow{(i,K,j):\overline{b}a} \overline{b}a[i,K].\overline{c}d \quad \text{and}$$
$$t_2 : \overline{b}a[i,K].\overline{c}d \xrightarrow{(i',K',j'):\overline{c}d} \overline{b}a[i,K].\overline{c}d[i',K']$$

In the process $X' = \overline{b}a[i,K].\overline{c}d[i',K']$, by using Definition 56, we have $i \sqsubset_{X'} i'$. Since key i identifies transition t_1 and key i' transition t_2, we have that $t_1 \sqsubset t_2$.

Example 28 If we take the resulting process from the example above and use it as initial state for the backward transition, i.e. we consider process $X = \overline{b}a[i,K].\overline{c}d[i',K']$ and backward transitions:

$$t_1 : X \rightsquigarrow^{(i',K',j'):\overline{c}d} \overline{b}a[i,K].\overline{c}d \quad \text{and}$$
$$t_2 : \overline{b}a[i,K].\overline{c}d \rightsquigarrow^{(i,K,j):\overline{b}a} \overline{b}a.\overline{c}d$$

In the process $X = \overline{b}a[i,K].\overline{c}d[i',K']$, by using Definition 56 for the backward transitions, we have $i \sqsubset_X i'$ what implies that $t_1 \sqsubset t_2$.

We define object causality directly on the transitions and use contextual cause set K to keep track of it.

Definition 58 (Object causality) *Transition* $t_1 : X \xrightarrow{(i_1,K_1,j_1):\alpha_1} X'$ *is an* object cause *of transition* $t_2 : X' \xrightarrow{(i_2,K_2,j_2):\alpha_2} X''$*, written* $t_1 < t_2$*, if* $i_1 \in K_2$ *or* $i_2 \in K_1$ *(for the backward transition) and* $t_1 \neq t_2$*.* Object causality*, denoted with* $\ll$*, is obtained by reflexive and transitive closure of* $<$*.*

Example 29 Consider the process $X = \nu a_\Delta(\overline{b}^* a^* \mid a^*(z))$ and computation:

$$\xrightarrow{(i_1,*,*):\overline{b}\langle \nu a_\Delta \rangle} \xrightarrow{(i_2,\{i_1\},*):a(z)}$$

We can notice that $i_1 \in K_2$, since $K_2 = \{i_1\}$ what implies that the second transition is caused by the first one.

Definition 59 (Causality relation and concurrency) *The* causality relation $\prec$ *is the reflexive and transitive closure of structural and object cause:* $\prec = (\sqsubseteq \cup \ll)^*$. *Two transitions are* concurrent *if they are not causally related.*

We now give some additional properties and definitions necessary to prove our main results, Square Lemma (Lemma 20) and causal-consistency (Theorem 10). We start by showing the following property stating that in a reachable reversible process all restriction νa_Δ of the same name a are nested.

Lemma 19 *If process* $X = C[\nu a_\Delta(Y) \mid Y']$ *is reachable, then* $\nu a_{\Delta'} \notin Y'$, *for all non-empty* Δ *and* Δ' ($\texttt{empty}(\Delta) = false$ *and* $\texttt{empty}(\Delta') = false$).

Proof The proof is by induction on the trace that leads to the process X: $X_1 \to \cdots \to X_n \to X$, where X_1 is an initial reversible process[1], and last applied rule on the transition $X_n \to X$. Base case is trivial, since for every $\nu a_\Delta \in X_1$, $\texttt{empty}(\Delta) = true$. In the inductive case, we have that in the transition $X_n \to X$, property holds for X_n. The proof continues by case analysis on the last applied rule on the transition $X_n \to X$. More details can be found in Appendix A. □

Before stating the permutation of concurrent transitions, we need to define an equivalence on the labels of the transitions.

Definition 60 (Label equivalence) *Label equivalence,* $=_\lambda$, *is the least equivalence relation satisfying:* $(i, K, j) : \overline{b}\langle \nu a_\Delta \rangle =_\lambda (i, K, j) : \overline{b}\langle \nu a_{\Delta'} \rangle$ *for all* i, j, K, a, b *and* $\Delta, \Delta' \subseteq \mathcal{K}$. *(Having an indexed set* Γ_w *for* Δ *we disregard index* w, *and observe* $\Gamma \subseteq \mathcal{K}$.*)*

The label equivalence is necessary since actions are bringing information about Δ into the labels. By permuting the transitions, content of Δ is changing. To illustrate this, let us consider the following example.

Example 30 Consider the process $X = \nu a_\emptyset(\overline{b}^* a^* \mid \overline{c}^* a^*)$ and the case when $\Delta = \Gamma$. For instance, process X can first execute output on the

[1]From Definition 53 we have that the reversible process X is initial when all its names are decorated with the $*$ and for all restrictions νa_Δ, for some name a, Δ is empty

channel b identified with the key i_1 and then output on the channel c with key i_2, and we have:

$$X \xrightarrow{(i_1,*,*):\overline{b}\langle \nu a_\emptyset \rangle} \nu a_{\{i_1\}}(\overline{b}^* a^*[i_1,*] \mid \overline{c}^* a^*)$$
$$\xrightarrow{(i_2,*,*):\overline{c}\langle \nu a_{\{i_1\}} \rangle} \nu a_{\{i_1,i_2\}}(\overline{b}^* a^*[i_1,*] \mid \overline{c}^* a^*[i_2,*]) = X_1$$

Now, if we execute actions in the opposite order, we have:

$$X \xrightarrow{(i_2,*,*):\overline{c}\langle \nu a_\emptyset \rangle} \nu a_{\{i_2\}}(\overline{b}^* a^* \mid \overline{c}^* a^*[i_2,*])$$
$$\xrightarrow{(i_1,*,*):\overline{b}\langle \nu a_{\{i_2\}} \rangle} \nu a_{\{i_1,i_2\}}(\overline{b}^* a^*[i_1,*] \mid \overline{c}^* a^*[i_2,*]) = X_2$$

We can notice that the resulting processes in both computations are the same, i.e. $X_1 = X_2$. In the labels of the transitions, we can see that $(i_1,*,*) : \overline{b}\langle \nu a_\emptyset \rangle =_\lambda (i_1,*,*) : \overline{b}\langle \nu a_{\{i_2\}} \rangle$, since the only difference is in the set Γ. Similar for the transitions on the channel c, we have $(i_2,*,*) : \overline{c}\langle \nu a_{\{i_1\}} \rangle =_\lambda (i_2,*,*) : \overline{c}\langle \nu a_\emptyset \rangle$.

Notation 5 From the fact that all restriction of the same name are nested (Lemma 19) and the design of the framework, we can write a reversible process X as: $X = \nu\widetilde{a_{n\Delta_n}} C_n[\ldots \nu\widetilde{a_{0\Delta_0}} C_0[X_1]]$, where contexts of the process that do not contain any restriction on a are represented with the $C_0, \ldots C_n$ and $\nu\widetilde{a_{l\Delta_l}} = \nu\widetilde{a_{l1\Delta_{l1}}} \ldots \nu\widetilde{a_{ln\Delta_{ln}}}$. For the sake of simplicity we shall assume that the vector of names $\nu\widetilde{a_{l\Delta_l}}$ is a singleton and write: $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$.

With the next properties, we show how in the transition $t : X \xrightarrow{(i,K,j):\alpha} X'$, executing action α influences the resulting process X'. Depending of the nature of action α, just one part of it or whole precess X' will be modified.

Property 4 *Given a process $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$, and transition $t : X \xrightarrow{(i,K,j):\alpha} X'$, where $\alpha \neq \tau$, executed by any component in X, there are two possibilities for modifying the resulting process X', depending on the nature of the action α:*

- *if $\alpha = \overline{b}\langle \nu a_\Delta \rangle$, for some name b, then transition t modifies the component on which it is executed and all the restrictions on the name a before it;*

- *if $\alpha \neq \overline{b}\langle \nu a_\Delta \rangle$, then transition t modifies only the component on which it is executed:*

Proof The proof follows directly from the semantics of the framework.□

For instance, consider the process $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$ and transition $t : X \xrightarrow{(i,K,j):\alpha} X'$, where $\alpha \neq \tau$. Let us assume that transition t is performed by the process X_1. Then we have:

- if $\alpha = \overline{b}\langle \nu a_\Delta \rangle$, for some name b; transition t modifies the process X_1 and all the restrictions, since all of them are before X_1, and we have:

$$t : X \xrightarrow{(i,K,j):\overline{b}\langle \nu a_{\Delta_n} \rangle} \nu a_{\Delta'_n} C_n[\ldots \nu a_{\Delta'_0} C_0[X'_1]]$$

  Every time when action α passes the restriction on the name a, rule OPEN is applied and key i is added to Δ.

- if $\alpha \neq \overline{b}\langle \nu a_\Delta \rangle$, then transition t modifies only the process X_1:

$$t : X \xrightarrow{(i,K,j):\alpha} \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X'_1]]$$

Property 5 *Given a process $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$, and transition $t : X \xrightarrow{(i,*,*):\tau} X'$, there are two possibilities for modifying the resulting process X', depending whether one or two contexts are involved into the transition t:*

- *if one context is included, then transition t modifies just that context*

- *if two contexts are involved and name a is used in the object position such that rules* CLOSE *and* CLOSE$^\bullet$ *can be applied, then transition t modifies both contexts with the first restrictions before them and restrictions on the name a between the contexts; otherwise, transition t modifies just the two contexts.*

Proof The proof is straightforward from the rules for communication.□

For example, given a process $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$, and transition $t : X \xrightarrow{(i,*,*):\tau} X'$, we have:

- if the communication happened in the process X_1, then

$$t : C[X_1] \xrightarrow{(i,*,*):\tau} C[X_1']$$

for some context C. We can notice that transition t modifies only the elements of the process X_1, not the context C, and we have:

$$t : \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]] \xrightarrow{(i,*,*):\tau} \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1']]$$

- if communication involves two contexts $C_i[\bullet]$, $C_j[\bullet]$ and name a is used in object position, rules CLOSE or CLOSE$^\bullet$ can be applied. We have the following transition:

$$t : \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\nu a_{\Delta_0} C_0[X_1]]]] \xrightarrow{(i,*,*):\tau}$$
$$\nu a_{\Delta_n} C_n[\ldots \nu a'_{\Delta_i'} C_i'[\ldots \nu a_{\Delta_j'} C_j'[\nu a_{\Delta_0} C_0[X_1]]]]$$

where the number of restrictions in $\nu a'_{\Delta_i'}$, depends on whether forward (rule CLOSE adds a restriction) or backward (rule CLOSE$^\bullet$ removes a restriction) rule is used.

Now, we have all necessary definitions to show the so-called Square Lemma stating that concurrent transitions of the framework can be permuted where permutations is allowed up to label equivalence.

Lemma 20 (Square Lemma) *If $t_1 : X \xrightarrow{\mu_1} Y$ and $t_2 : Y \xrightarrow{\mu_2} Z$ are two concurrent transitions, there exist $t_2' : X \xrightarrow{\mu_2'} Y_1$ and $t_1' : Y_1 \xrightarrow{\mu_1'} Z$ where $\mu_i =_\lambda \mu_i'$.*

Proof The proof is by case analysis on the form of the transitions t_1 and t_2. We shall consider four main cases on whether transitions t_1 and t_2 are synchronisations or not and then proceed with induction on the structure of the process while checking all possible combination of the rules applied on the transitions t_1 and t_2. More details about the proof, can be found in Appendix A. □

We write $t_2 = t_2'/t_1$ for a residual of t_2' after t_1. Additionally, we bring from the π-calculus standard notions on the transitions (Definition 25). Two transitions that have the same source, are called *coinitial*; if they have

the same target, they are *cofinal* and if target of one is source of the other transition, they are called *composable*. We denote with $t_1; t_2$ a sequence of pairwise composable transitions that is called *trace*. *Empty* trace is written as ϵ.

Now we revise the definition of *equivalence up-to permutation* introduced in [23]. It is an adaptation of equivalence between traces introduced in [15; 55] which additionally removes from the trace a transitions triggered in both directions. It basically states that concurrent actions can be permuted and that trace composed by transition and its inverse is equivalent to the empty trace. Formally:

Definition 61 (Equivalence up-to permutation) *Equivalence up-to permutation, $\sim$, is the least equivalence relation on the traces, satisfying:*

$$t_1; (t_2/t_1) \sim t_2; (t_1/t_2) \qquad t; t^{\bullet} \sim \epsilon$$

In the following we give some properties defined on the transitions, necessary to show the main result of this section: causal-consistency of the framework.

Definition 62 *Two transitions t_1 and t_2 are* prefix equivalent, *written $t_1 =_p t_2$ if they add or remove the same past element $\pi[i, K]$ from the history context of a process.*

We would like to remind that history context is built from the past prefixes. For example, in the process $\overline{a}^* b^*[i, K].c^*(x).\mathbf{P}$, the history context is $\mathtt{H} = \overline{a}^* b^*[i, K].\bullet$, while in the process $\overline{a}^* b^*.c^*(x).\mathbf{P}$, the history context is empty (i.e. $\mathtt{H} = \bullet$).

Example 31 *Given the process $\overline{a}^* b^*.P_1 \mid \overline{a}^* b^*.P_2$, we have two transitions*

$$t_1 : \overline{a}^* b^*.P_1 \mid \overline{a}^* b^*.P_2 \xrightarrow{(i,*,*):\overline{a}b} \overline{a}^* b^*[i, *].P_1 \mid \overline{a}^* b^*.P_2$$
$$t_2 : \overline{a}^* b^*.P_1 \mid \overline{a}^* b^*.P_2 \xrightarrow{(i,*,*):\overline{a}b} \overline{a}^* b^*.P_1 \mid \overline{a}^* b^*[i, *].P_2$$

Transitions t_1 and t_2 are not the same, but they are prefix equivalent because they add the same past element into the history. The LTS ensures that keys are unique in the process.

Lemma 21 *If transitions t_1 and t_2 are prefix equivalent, coinitial and they are on exactly the same prefix in a process, then $t_1 = t_2$.*

Proof The proof follows from the fact that keys are unique and that transitions are coinitial on the same prefix. □

The following lemma states that reversible computation can be rearranged as the backward-only transitions, followed by the forward-only one.

Lemma 22 (Parabolic traces) *Let s be a trace. Then there exist a backward-only trace r and a forward-only trace r' such that $s \sim r; r'$.*

Proof The proof is by induction on the length of s and the distance between the very first transition in s and the pair of transitions contradicting the statement of the lemma. More details can be found in Appendix A.□

With the next lemma we show that if s_1 and s_2 are two coinitial and cofinal traces and s_2 is made just of the forward transitions, then exist the trace s_1' forward-only equivalent to the s_1. Intuitively, backward transitions of the trace s_1 can be deleted, since s_1 and s_2 are coinitial and cofinal and s_2 is forward-only.

Lemma 23 *Let us denote with s_1 and s_2 two coinitial and cofinal traces, where s_2 is forward only. Then there exists a forward-only trace s_1', shorter or equal to s_1, such that $s_1 \sim s_1'$.*

Proof The proof is by induction on the length of s_1. Full proof can be found in Appendix A. □

Now we can show the main result of this Section stating that reversibility in our framework is causally-consistent. In the other words, we prove that while executing backward actions in the framework, causality is respected.

Theorem 10 (Causal-consistency) *Two traces are coinitial and cofinal if and only if they are equivalent up-to permutation.*

Proof Let us denote two traces with s_1 and s_2. If $s_1 \sim s_2$ then from the definition of $\sim$ (Definition 61) we can conclude that they are coinitial and cofinal.

Let us suppose that s_1 and s_2 are coinitial and cofinal. From the Lemma 22 we can suppose that they are parabolic. We shall reason by induction on the lengths of s_1, s_2 and on the depth of the very first disagreement between them. Full proof can be found in Appendix A. □

5.6 Correspondence with Boreale and Sangiorgi's semantics

In this Section we prove a causal correspondence between Boreale and Sangiorgi's late semantics, revised in Section 4.1, and our framework when data structure Δ is instantiated with the indexed set Γ_w.

We first point out the difference on structural causality in both settings by looking at the traces of processes.

Since the framework is meant for reversible computation (Lemma 18: every forward action can be undone), every action has its corresponding key, including τ-actions, what is not the case in [14], where synchronisation only merges cause sets of the actions that communicate. Additionally, for the same reason, the framework keeps track of every action that was executed, while semantics in [14], just records sets of the causes that trigger the performing action. We give the following example to clarify it.

Example 32 Let us consider the π-calculus process $P = \overline{b}a.\overline{c}d \mid \overline{e}a_1.c(x).Q$. By executing two output actions on the channel b and e and synchronisation on the channel c, we obtain:

- in [14], the resulting process is $A = \{i_1, i_2\} :: \mathbf{0} \mid \{i_1, i_2\} :: Q$ where keys i_1 and i_2 correspond to actions $\overline{b}a$ and $\overline{e}a_1$, respectively. We can notice that τ-action just merged the cause sets $\{i_1\}$ and $\{i_2\}$. For more details about the computation, see Example 13.

- in our framework, the resulting process is

$$X = \overline{b}a[i_1, *].\overline{c}d[i_3, *] \mid \overline{e}a_1[i_2, *].c(x)[i_3, *].Q$$

  As we can notice, τ-action is identified by the key i_3.

Another difference is that in [14], the executing action brings its cause set into the label of the transition, while in the framework, structural cause set is defined in the resulting process of the transition. In order to better understand the difference, we give one simple example.

Example 33 Consider the π-calculus process $P = \overline{b}a.\overline{c}d.\overline{e}f$:

- in [14], actions $\overline{b}a$ with i_1 and $\overline{c}d$ with i_2 can be performed and we obtain the causal process $A = \{i_1, i_2\} :: \overline{e}f$. The transition for the action $\overline{e}f$ is

$$\{i_1, i_2\} :: \overline{c}d \xrightarrow[\{i_1,i_2\}]{i_3:\overline{e}f} \{i_1, i_2, i_3\} :: \mathbf{0}$$

  We can notice that in the label of the transition we can see the whole set of the causes that cause action $\overline{e}f$.

- in the framework, the same actions are performed and obtained reversible process is $X = \overline{b}a[i_1, *].\overline{c}d[i_2, *].\overline{e}f$. The transition for the action $\overline{e}f$ is

$$\overline{b}a[i_1, *].\overline{c}d[i_2, *].\overline{e}f \xrightarrow{(i_3,*,*):\overline{e}f}\!\!\!\!\rightarrow \overline{b}a[i_1, *].\overline{c}d[i_2, *].\overline{e}f[i_3, *] = X'$$

  The structural causality is defined on prefixes of the resulting process X' (Definition 56), hence the set of the structural causes of the action with the key i_3 can be computed after the execution.

Notation 6 To distinguish labels of two semantics, we write:

- transition from [14] as $A \xrightarrow[\mathtt{K}]{\zeta} A_1$, where

$$\zeta = i : \beta \text{ and } \beta = \overline{b}a \mid b(x) \mid \overline{b}\langle \nu a \rangle \mid \tau$$

  and $i \in \mathcal{K}$ ($\mathcal{K}$ is infinite denumerable set of keys);

- transition from the framework as $X \xrightarrow{\mu}\!\!\!\!\rightarrow X_1^{K_F}$, where $\mu = (i, K, j) : \alpha$ and $\alpha = \overline{b}a \mid b(x) \mid \overline{b}\langle \nu a_\Delta \rangle \mid \tau$ with $i \in \mathcal{K}$, $j \in \mathcal{K}_*$, $K \subset \mathcal{K}_*$ and K_F is the set of the keys belonging to the actions that structurally cause the action μ.

Now we give the definition of the function $\gamma(\cdot)$ that translate the label from the framework μ into a label from Boreale and Sangiorgi's semantics ζ:

Definition 63 *The function γ that maps label from the framework μ, with a label from [14] ζ, is inductively defined as follows:*

$$\begin{array}{lll}
\gamma((i,K,j):\alpha) = i:\gamma(\alpha) & \textit{when } \alpha \neq \tau & \gamma((i,*,*):\tau) = \tau \\
\gamma(\overline{b}\langle \nu a_{\Delta}\rangle) = \overline{b}\langle \nu a\rangle & \textit{when } \texttt{empty}(\Delta) = true & \gamma(b(x)) = b(x) \\
\gamma(\overline{b}\langle \nu a_{\Delta}\rangle) = \overline{b}a & \textit{when } \texttt{empty}(\Delta) = false & \gamma(\overline{b}a) = \overline{b}a
\end{array}$$

Focusing on structural causality, the main difference between two semantics is in the τ-actions, as illustrated with Example 32. Therefore, we need to provide the connection between structural cause sets $\texttt{K}$ and K_F of these two semantics. The idea is to represent structural dependences between keys in the reversible process X as a directed graph (digraph) and by removing the nodes (keys) belonging to τ-actions, obtain the cause set $\texttt{K}$ of the corresponding causal process A.

Before showing the example that illustrates our method, we give a basic notions about the graphs [36]. A *directed graph* or *digraph* $G = (V, E)$ consists of the non-empty set of vertices V (nodes) and the set of directed edges $E = \{(v_1, v_2)|$ where $v_1, v_2 \in V\}$. In the edge (v_1, v_2), v_1 is a *source* vertex of the edge, while v_2 is a *target* vertex.

In order to illustrate the method which connects structural cause sets $\texttt{K}$ and K_F, we give the following example.

Example 34 Let us consider the π-calculus process

$$P = \overline{b}a.\overline{c}d \mid \overline{b_1}a_1.c(x).\overline{b}c.\overline{b_2}a_2 \mid \overline{f}e.b(y)$$

where the actions $\overline{b}a, \overline{b_1}a_1, \tau_c, \overline{f}e, \tau_b$ and $\overline{b_2}a_2$ are identified with the keys i_1, i_2, i_3, i_4, i_5 and i_6, respectively.

- in [14] semantics, the resulting process is

  $$A = \{i_1, i_2\} :: \mathbf{0} \mid \{i_1, i_2, i_4\} :: \{i_6\} :: \mathbf{0} \mid \{i_1, i_2, i_4\} :: \mathbf{0}$$

  As we can notice, τ-actions do not have keys, hence i_3, i_5 are not in A. We separated set $\{i_6\}$ from the rest of the cause set to emphasise the fact that action with key i_6 depends on the cause set $\texttt{K} = \{i_1, i_2, i_4\}$.

- in the resulting process of the framework, we are interested just in the keys, not in the executed actions, hence, for the sake of simplicity,

we omit everything in the history of the process, except keys. The result of the computation above is $i_1.i_3.\mathbf{0} \mid i_2.i_3.i_5.i_6.\mathbf{0} \mid i_4.i_5.\mathbf{0}$. We can represent dependences between the keys as digraph $G = (V, E)$, given bellow:

$$\begin{tikzcd}
i_1 \arrow[r] & i_3 \arrow[d, leftrightarrow] & & \\
i_2 \arrow[r] & i_3 \arrow[dr] & & \\
& & i_5 \arrow[r] \arrow[d, leftrightarrow] & i_6 \\
& i_4 \arrow[r] & i_5 &
\end{tikzcd}$$

In the digraph G, nodes represents keys and directed edge $i \to i'$ means that key i causes key i'. From the graph, we can notice that cause set of the action with key i_6 is $K_F = \{i_1, i_2, i_3, i_4, i_5\}$. If we remove all bidirectional edges, join the nodes that they connect (keys belong to synchronisations) and rename them into τ_i, we obtain the subgraph $G' = (V', E')$:

$$\begin{tikzcd}
i_1 \arrow[dr] & & & \\
& \tau_2 \arrow[dr] & & \\
i_2 \arrow[ur] & & \tau_1 \arrow[r] & i_6 \\
& i_4 \arrow[ur] & &
\end{tikzcd}$$

Now, if we take the set of vertices V' and remove all τ_i nodes, we obtain the cause set $\mathtt{K}$ of the action i_6 in Boreale and Sangiorgi's semantics ($\mathtt{K} = V' \setminus \{\tau_i\}$).

The whole algorithm of connecting sets K_F and $\mathtt{K}$, illustrated in example 34 is called *Removing Keys from a Set* written as $\mathtt{Rem}(K_F) = \mathtt{K}$. In the further text we formally define the method $\mathtt{Rem}$.

Suppose that we have two transitions:

$$t : X \xrightarrow{\mu} X_1^{K_F} \quad \text{and} \quad t' : A \xrightarrow[\mathtt{K}]{\zeta} A_1$$

with $\gamma(\mu) = \zeta$ and processes X and A, translated in π-calculus, give the same process P, i.e. $\varphi(X) = \lambda(A) = P$. The same holds fo the processes $X_1^{K_F}$ and A_1, i.e., we have$\varphi(X_1^{K_F}) = \lambda(A_1) = Q$. The function $\varphi(\cdot)$

is the same as erasing function from Definition 54 with additional rule $\varphi(X^{K_F}) = \varphi(X)$ that removes the set K_F from the reversible process. The function λ that translates causal term into π-calculus process is defined as:

Definition 64 *The erasing function λ that maps causal processes from Boreale and Sangiorgi's semantics to the π-calculus is inductively defined as follows:*

$$\lambda(A \mid A') = \lambda(A) \mid \lambda(A') \qquad \lambda(\mathtt{K} :: A) = \lambda(A)$$
$$\lambda(\nu a(A)) = \nu a(\lambda(A)) \qquad \lambda(P) = P$$

The structural dependences between the past prefixes belonging to the history of the process X involved into the execution of the action $\alpha \in t, t'$ can be represented with digraph in the following way: keys belonging to the past prefixes are represented as vertices of the digraph (the same keys which are representing synchronisation, are represented by two vertices with the same name); structural dependences between the keys are represented by directed edges where between the same vertices we shall have edges in both directions. Formally, we have:

Definition 65 *Given a transition $t : X \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow X_1^{K_F}$ the structural dependences between the past prefixes involved in the execution of the action α contained in the history of the reversible process $X_1^{K_F}$ can be represented as a digraph $G = (V, E)$, in the following way:*

- $\forall\, \pi[i_1, K] \in X_1^{K_F} \wedge i_1 \sqsubseteq_{X_1^{K_F}} i \implies i \in V$
- $\forall\, i_1, i_2 \in V$ *such that* $\pi[i_1, K].\pi'[i_2, K'] \in X_1^{K_F} \implies (i_1, i_2) \in E'$
- $E = E' \cup \{(i_1, i_2) |$ *when* $i_1, i_2 \in V \wedge i_1 = i_2\}$
  $\cup \{(i_2, i_1) |$ *when* $i_1, i_2 \in V \wedge i_2 = i_1\}$

where V is a multiset of vertices and E is a set of directed edges. Having a digraph $G = (V, E)$, structural cause set of the action α is $K_F = V \setminus \{i\}$.

Since bidirectional edges represent dependency flow between vertices with the same name, we can remove them and join two vertices into one, renamed to τ. This operation is known as edge contraction [36]. Here we adapt it to bidirectional edges as follows:

Definition 66 (Bidirectional edge contraction) *Bidirectional edge contraction is an operation defined on the directed graph $G = (V, E)$, as follows:*

- $E' = E \setminus ((i_1, i_2) \cup (i_2, i_1))$ *when* $i_1 = i_2$
- $V' = (V \setminus \{i_1, i_2\}) \cup \{\tau\}$
- $\forall (i, i_l), (i_l, i) \in E$ *where* $l \in \{1, 2\}$, *we have that* $(i, \tau), (\tau, i) \in E'$,

where $G' = (V', E')$ is the obtained subgraph.

In words, the above definition removes bidirectional edge and substitute two nodes that it connects with the τ node. Additionally, all the edges that have source or target in the removed nodes will have source or target in τ node. For instance, let $G = (V, E)$ be directed graph and G' be the subgraph obtained from the graph G by applying Definition 66:

$$G: \quad \begin{array}{ccccc} i_1 & \longrightarrow & i_3 & \longrightarrow & i_4 \\ & & \updownarrow & & \\ i_2 & \longrightarrow & i_3 & & \end{array} \qquad G': \quad \begin{array}{ccccc} i_1 & \searrow & & & i_4 \\ & & \tau & \nearrow & \\ i_2 & \nearrow & & & \end{array}$$

From the representations of the graphs G and G' above, we can notice that nodes labeled with i_3 together with the bidirectional edge, are substituted with the node τ. At the same time, edge (i_1, i_3) becomes (i_1, τ), and similar for the rest of edges containing nodes i_3.

By applying bidirectional edge contraction (Definition 66) on every bidirectional edge of a graph $G = (V, E)$, we obtain a subgraph $G' = (V', E')$ in which all pairs of the same vertices are joined and renamed as τ_l, for $l = 1, 2, ...$ Set V' differs from the multiset V in having τ_l vertices instead of the pairs of vertices labeled with the same name (originally belonging to silent moves in the framework). Hence, we can conclude that $\mathtt{K} = V' \setminus (\{i\} \cup \tau_l)$.

The method 'Removing Keys from a Set', denoted as `Rem` is formally defined as.

Definition 67 (Method `Rem`) *Given a two transitions $t : X \overset{\mu}{\twoheadrightarrow} X_1^{K_F}$ with $\mu = (i, K, j) : \alpha$ and $t' : A \underset{\mathtt{K}}{\overset{\zeta}{\rightarrow}} A_1$, where $\gamma(\mu) = \zeta$ and $\varphi(X) = \lambda(A) = P$ and $\varphi(X_1^{K_F}) = \lambda(A_1) = Q$, correspondence between sets K_F and $\mathtt{K}$ is defined through method `Rem`, given with following steps:*

- *structural dependences in the process $X_1^{K_F}$ involved in the transition t are represented as digraph $G = (V, E)$ (Definition 65), where $K_F = V \setminus \{i\}$;*

- *by applying Definition 66 on every bidirectional edge in $G = (V, E)$, the subgraph $G' = (V', E')$ is obtained, where $V' \setminus (\{i\} \cup \tau_l) = \mathtt{K}$ (τ_l represents all the nodes obtained by bidirectional edge contraction)*

Now we have all auxiliary definitions and lemmata necessary to prove the structural correspondence between two causal semantics.

Lemma 24 (Structural correspondence) *Starting from initial π-calculus process P, where $P = \lambda(A_1) = \varphi(X_1)$, we have:*

1. *if $A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ is a trace in causal semantics [14], then exists a trace $X_1 \xrightarrow{\mu_1} X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n} X_{n+1}^{K_{Fn}}$ and K_{Fi} in the framework, such that for all i, $\lambda(A_i) = \varphi(X_i^{K_{Fi}})$, $\zeta_i = \gamma(\mu_i)$ and $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$, for $i = 1, ..., n$.*

2. *if $X_1 \xrightarrow{\mu_1} X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n} X_{n+1}^{K_{Fn}}$ is a trace in the framework, then exists a trace $A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ in causal semantics, where for all i, $\lambda(A_i) = \varphi(X_i^{K_{Fi}})$, $\zeta_i = \gamma(\mu_i)$ and $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$, for $i = 1, ..., n$.*

Proof Both directions (*1.* and *2.*) are proved by induction on the length of the computation followed by induction on the structure of the π-calculus process P and last applied rule on the transition t, where $t : A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ and $t' : X_n^{K_{Fn-1}} \xrightarrow{\mu_n} X_{n+1}^{K_{Fn}}$. Full proof is given in Appendix B. □

The object causality in Boreale and Sangiorgi's semantics is defined on the trace of a process (Definition 28). The first action that extrudes a bound name will cause all the future actions using that name in any position of the label.

We can define object causality induced by the framework, on the forward trace of a reversible process. Previously it was defined on two consecutive transitions (Definition 58).

Definition 68 (Object causality on the trace in the framework) *In the trace* $t_1 : X_1 \xrightarrow{(i_1,K_1,j_1):\alpha_1} X_2 \cdots t_n : X_n \xrightarrow{(i_n,K_n,j_1):\alpha_n} X_{n+1}$*, transition* t_h *is an* object cause *of transition* t_l*, written* $1 \leq t_h < t_l \leq n$*, if* $i_h \in K_l$

The next theorem will prove causal correspondence between causality in the framework when memory Δ is instantiated with Γ_w and Boreale and Sangiorgi's late causal semantics.

Theorem 11 (Causal correspondence) *The reflexive and transitive closure of causality introduced in [14] coincides with the causality of the framework when* $\Delta = \Gamma_w$*.*

Proof The proof relies on Lemma 24 and the fact that object dependence induced by input action in Boreale and Sangiorgi's semantics is subject dependence as well. By design of the framework and definitions for predicates $\texttt{Cause}(\cdot)$ and $\texttt{Update}(\cdot)$ the first extrusion of a name will cause every other action using that name (this is accomplished with the rules OPEN and CAUSE REF). In Definition 28 object dependence induced by an input action is also the structural one, and the one induced by extrusion coincides with object dependence in the framework.

5.7 Causal Bisimulation

In this Section we give a brief description of the *causal bisimulation* defined on the framework. We introduce the notion of causal bisimulation which abstracts away from the notion of causality used, and does not distinguish object from the subject causality. Additionally, bisimulation distinguishes forward steps from backward steps, since mixing them leads to an equivalence which is coarser than weak bisimulation (see [51]).

In the following, we define when two transition are caused by the same actions in the past. We recall that set of the structural causes K_t of the transition $t : X \xrightarrow{(i,K,j):\alpha} X_1^{K_t}$ is defined as $K_t = \{i' \in \texttt{key}(X_1) | i' \sqsubseteq_{X_1} i\}$. Then, two transitions t_1 and t_2 are caused by the same actions in the past if:

Definition 69 *Two transitions* $t_1 : X \xrightarrow{(i,K_1,j):\alpha} X_1^{K_{t_1}}$ *and* $t_2 : Y \xrightarrow{(i,K_2,j):\alpha} Y_1^{K_{t_2}}$ *where* K_{t_1} *and* K_{t_2} *are sets of the structural causes of the transitions* t_1 *and* t_2*, are caused by the same actions in the past if* $K_{t_1} \cup K_1 = K_{t_2} \cup K_2$*.*

Now we give the definition of the causal bisimulation which do not distinguish between two types of dependences (object and the subject one). We define it for the forward and for the backward transitions.

Definition 70 (Causal Bisimulation) *A symmetric relation* $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{X}$ *is a* causal bisimulation *if* $X\mathcal{R}Y$ *implies:*

(i) *if* $t_1 : X \xrightarrow{(i,K_1,j):\alpha}\!\!\!\!\rightarrow X'$ *then there exist* Y' *and* t_2 *such that* $t_2 : Y \xrightarrow{(i,K_2,j):\alpha}\!\!\!\!\rightarrow Y'$ *and* $X'\mathcal{R}Y'$*, and the two transitions are caused by the same actions in the past;*

(ii) *if* $t_1 : X \rightsquigarrow^{(i,K_1,j):\alpha} X'$ *then there exist* Y' *and* t_2 *such that* $t_2 : Y \rightsquigarrow^{(i,K_2,j):\alpha} Y'$ *and* $X'\mathcal{R}Y'$*, and the two transitions are caused by the same action in the past.*

Two reversible processes X *and* Y *are* causally bisimilar*, written* $X \approx Y$*, if* $X\mathcal{R}Y$ *for some causal bisimulation* $\mathcal{R}$*.*

Let us note that because of communication keys (and other causal information) two τ actions are also considered different if they refer to two different synchronisations. We now show how by using different notions of causality (and semantics) two processes can be considered causally bisimilar or not.

Example 35 Let us consider reversible processes:

$$X = \nu a_\emptyset(\overline{b}^* a^*.\overline{c}^* a^* + \overline{c}^* a^*.\overline{b}^* a^*) \quad \text{and} \quad Y = \nu a_\emptyset(\overline{b}^* a^* \mid \overline{c}^* a^*)$$

These two processes are not causally bisimilar if we consider $R\pi$ causality or [19], but they are if we consider the causal semantics given in [14].

In $R\pi$ causality, after executing the action $\overline{b}a$ with the key i we obtain the processes

$$X' = \nu a_{\{i\}}(\overline{b}^* a^*[i,*].\overline{c}^* a^* + \overline{c}^* a^*.\overline{b}^* a^*) \text{ and } Y' = \nu a_{\{i\}}(\overline{b}^* a^*[i,*] \mid \overline{c}^* a^*)$$

The execution of the action $\overline{c}a$ in the process X' has structural dependency on the first action, while in the process Y' it is concurrent with the action

$\overline{b}a$. Hence processes X and Y are not causally bisimilar. Similar for the semantics in [19].

In causal notion induced by [14], after executing the action $\overline{b}a$ with key i we obtain the processes $X' = \nu a_{\{i\}_i}(\overline{b}^* a^*[i,*].\overline{c}^* a^* + \overline{c}^* a^*.\overline{b}^* a^*)$ and $Y' = \nu a_{\{i\}_i}(\overline{b}^* a^*[i,*] \mid \overline{c}^* a^*)$. The action $\overline{b}a$ is the very first action that extruded name a. The execution of the action $\overline{c}a$ with the key i' in the process X' has structural and object dependency on the first action, and in the process Y' has object dependency on the first action. Therefore, resulting processes are: $X'' = \nu a_{\{i,i'\}_i}(\overline{b}^* a^*[i,*].\overline{c}^* a^*[i',\{*,i'\}] + \overline{c}^* a^*.\overline{b}^* a^*)$ and $Y' = \nu a_{\{i,i'\}_i}(\overline{b}^* a^*[i,*]. \mid \overline{c}^* a^*[i',\{*,i'\}])$. Hence, in both processes action $\overline{c}a$ is caused by the first action and we have that X and Y are causally bisimilar.

Chapter 6

Conclusion and Future work

In this thesis we studied the expressiveness of the causal-consistent reversibility in the CCS [60] and π-calculus [73].

We have shown that the LTSs of two main forms of reversible CCS, namely RCCS [23] and CCSK [70], are isomorphic, hence they are different syntactic representations for the same behaviors. An explanation of this result is the existence of one causality notion in CCS. Nevertheless, the syntactic differences have an impact on their possible uses and extensions. The proof of the isomorphism relies on two encodings one of CCSK into RCCS and another one of RCCS into CCSK. The interesting fact about encodings is that non of them is uniform. Moreover, we have shown that no uniform encoding can exist since reachable RCCS processes are not closed under parallel composition.

Moving to the π-calculus, dependences between the actions can be caused by the structure of the process (structural dependence) or by extruding a name (object dependence). Different interpretations of the object dependence, give rise to many causal semantics for π-calculus. We mainly study three of them [14; 19; 20] representing three different approaches to causality in π-calculus. For that purpose, we devise a framework for reversible π-calculi, parametric with respect to the data

structure that stores information about the extrusions of a name. Different approaches to causality in π-calculus can be obtained by using a different data structures. We proved that reversibility introduced by the framework is causally-consistent and show causal correspondence between causal semantics given in [14] and corresponding instance of the framework. Additionally, we give the idea of the causal bisimulation that can be defined on the framework as a starting point in developing the behavioural theory of the framework.

Future works. The number of concurrent reversible calculi and languages is increasing, but there is very little literature on the relations between them. We are only aware of [53], where a classification of the different approaches is presented. However, the classification is just at a descriptive level. Hence, the problem of understanding the relations between different approaches stays open, and we plan to further investigate it in future work. We note that the approach of CCSK is very close to event transition systems [16], which are related also to some classes of event structures and of Petri Nets. Further analysis of this connection may give insight on the classes of calculi to which the CCSK approach can be applied and the ones to which it cannot be applied.

Regarding to π-calculus, as a future work we plan to prove causal correspondence with the semantics [19; 20] and to continue working towards a more parametric framework and to compare it with [37; 63]. Moreover it would be interesting to implement our framework in the psi-calculi framework [10], and to develop further behavioural theory of our framework and show how the framework can be used to study different notions of causality [32].

Appendix A

In this Section we give the detailed proofs from Section 5.5.2.

Lemma 19. If process $X = C[\nu a_\Delta(Y) \mid Y']$ is reachable, then $\nu a_{\Delta'} \notin Y'$, for all non-empty Δ and Δ' ($\texttt{empty}(\Delta) = false$ and $\texttt{empty}(\Delta') = false$).

Proof The proof is by induction on the trace that leads to the process X: $X_1 \to \cdots \to X_n \to X$, where X_1 is an initial reversible process[1], and last applied rule on the transition $X_n \to X$. Base case is trivial, since for every $\nu a_\Delta \in X_1$, $\texttt{empty}(\Delta) = true$. In the inductive case, we have that in the transition $X_n \to X$, property holds for X_n. We continue by case analysis on the last applied rule on the transition $X_n \to X$:

- rule PAR

$$\frac{Y_0 \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow Y_1 \quad i \notin Y' \quad \texttt{bound}(\alpha) \cap \texttt{free}(Y') = \emptyset}{Y_0 \mid Y' \xrightarrow{(i,K,j):\alpha}\!\!\!\!\rightarrow Y_1 \mid Y'}$$

  where property holds for $Y_0 \mid Y'$. We proceed with the following cases:

  - if $\nu a_\Delta \in Y_0$, then by inductive hypothesis $\nu a_{\Delta'} \notin Y'$. After the execution of the action α, property is preserved and it holds in $Y_1 \mid Y'$ as well.

[1]From Definition 53 we have that the reversible process X is initial when all its names are decorated with the $*$ and for all restrictions νa_Δ, for some name a, Δ is empty

- if $\nu a_\Delta \in Y'$, then by inductive hypothesis $\nu a_{\Delta'} \notin Y_0$, when Δ, Δ' are not empty (name a is free in Y', since Δ is not empty). To satisfy the property, we need to show that $\nu a_{\Delta'} \notin Y_1$ holds. Let us suppose the opposite, that $\nu a_{\Delta'} \in Y_1$. Then some restriction $\nu a_{\Delta''}$ needs to belong to Y_0 and the only possibility is $\nu a_{\Delta''} \in Y_0$ when $\texttt{empty}(\Delta'') = true$. In that case, action $\alpha = \overline{b}\langle \nu a_{\Delta''} \rangle$, where $\texttt{empty}(\Delta'') = true$, is executed, what is in the contradiction with the side condition in the rule PAR, since bound action a is between the free names in Y'. Hence, $\nu a_{\Delta'} \notin Y_1$ and property holds.

- rule COM

$$\frac{Y_0 \xrightarrow{(i,K,j):\overline{b}c}\!\!\!\!\rightarrow Y_1 \quad Y_0' \xrightarrow{(i,K',j'):b(x)}\!\!\!\!\rightarrow Y_1' \quad K =_* j' \ \wedge \ K' =_* j}{Y_0 \mid Y_0' \xrightarrow{(i,*,*):\tau}\!\!\!\!\rightarrow Y_1 \mid Y_1'\{^{c^i}/_x\}}$$

  where property holds for $Y_0 \mid Y_0'$. We proceed with the following cases:

  - if $\nu a_\Delta \in Y_0$, then by inductive hypothesis $\nu a_{\Delta'} \notin Y_0'$, when Δ, Δ' are not empty. We need to show that $\nu a_{\Delta'} \notin Y_1'\{^{c^i}/_x\}$.
    If $c = a$, then by design of the framework, action $\overline{b}c$ should be $\overline{b}\langle \nu a_{\Delta''} \rangle$, for some Δ'' and rule CLOSE should be applied.
    If $c \neq a$, then $\nu a_{\Delta'} \notin Y_1'$ since a is not in the object position of the output action $\overline{b}c$. Hence, for the process $Y_1 \mid Y_1'\{^{c^i}/_x\}$ property holds
  - if $\nu a_\Delta \in Y_0'$, then by inductive hypothesis $\nu a_{\Delta'} \notin Y_0$, when Δ, Δ' are not empty. Since the action $\overline{b}c$ is executed on the Y_0, there is no possibility for $\nu a_{\Delta'}$ to belong to the process Y_1, and we have $\nu a_{\Delta'} \notin Y_1$ and property holds.

For the rest of the rules, property trivially holds. □

Lemma 20. (**Square Lemma**) If $t_1 : X \xrightarrow{\mu_1} Y$ and $t_2 : Y \xrightarrow{\mu_2} Z$ are two concurrent transitions, there exist $t_2' : X \xrightarrow{\mu_2'} Y_1$ and $t_1' : Y_1 \xrightarrow{\mu_1'} Z$ where $\mu_i =_\lambda \mu_i'$.

Proof The proof is by case analysis on the form of the transitions t_1 and t_2. We shall consider four main cases on whether transitions t_1 and t_2 are synchronisations or not and then proceed with induction on the structure of the process while checking all possible combination of the rules applied on the transitions t_1 and t_2. We show just interesting cases when the restriction of a name is involved and reversible process is written in the form $X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_0} C_0[X_1]]$ (Notation 5).

We proceed with case analysis on whether transitions t_1 and t_2 are synchronisations or not:

1. t_1 and t_2 are not synchronisations. We need to prove that by changing the order of the transitions we shall obtain the same process. We consider the cases where transitions t_1 and t_2 modify not just their own context, but also other contexts or restrictions.

   Let us assume that transition t_1 modifies process X_1 and that performed action is $\alpha_1 = \overline{b}\langle \nu a_\Delta \rangle$. By the Property 4,we have

   $$t_1 : X \xrightarrow{\mu_1} \nu a_{\Delta'_n} C_n[\ldots \nu a_{\Delta'_0} C_0[X'_1]]$$

   where $\mu_1 = (i_1, K_1, j_1) : \overline{b}\langle \nu a_\Delta \rangle$. Let α_2 be the action of the transition t_2. If $a \notin \alpha_2$ then t_2 modifies just its own context (Property 4) and it is not prevented by the restrictions on a. Let us consider the case when $a \in \alpha_2$. We continue the proof with the induction on the structure of the process.

   - The base case of induction is to prove that if

     $$\nu a_\Delta (X_1 \mid X_2) \xrightarrow{\mu_1} \nu a_{\Delta'_1} (X'_1 \mid X_2) \xrightarrow{\mu_2} \nu a_{\Delta'} (X'_1 \mid X'_2)$$

     then

     $$\nu a_\Delta (X_1 \mid X_2) \xrightarrow{\mu'_2} \nu a_{\Delta'_2} (X_1 \mid X'_2) \xrightarrow{\mu'_1} \nu a_{\Delta'} (X'_1 \mid X'_2)$$

     Since t_1 has performed action $\alpha_1 = \overline{b}\langle \nu a_\Delta \rangle$, rules OPEN and OPEN$^\bullet$ can be used. We consider just the interesting cases when on t_1 is applied rule OPEN and continue with the case analysis on the rules that can be applied on t_2 such that $a \in \alpha_2$.

– Rule OPEN applied on t_2. We have

$$\nu a_\Delta(X_1 \mid X_2) \xrightarrow{(i_1,K_1,j_1):\overline{b}\langle\nu a_\Delta\rangle} \nu a_{\Delta+i_1}(X_1' \mid X_2)$$
$$\xrightarrow{(i_2,K_2,j_2):\overline{c}\langle\nu a_{\Delta+i_1}\rangle} \nu a_{\Delta+i_1+i_2}(X_1' \mid X_2')$$

where $X_2 \xrightarrow{(i_2,K,j_2):\overline{c}\langle\nu a_{\Delta'}\rangle} X_2'$. If $i_1 \in K_2$ then transition t_1 cause transition t_2 and this is not the case (they are concurrent). If $i_1 \in K$, then by definition of the predicate $\texttt{Update}(\cdot)$ on the rule OPEN, we will have $i_1 \in K_2$. Therefore, $i_1 \notin K, K_2$ and we can safely commute transitions and obtain:

$$\nu a_\Delta(X_1 \mid X_2) \xrightarrow{(i_2,K_2,j_2):\overline{c}\langle\nu a_\Delta\rangle} \nu a_{\Delta+i_2}(X_1 \mid X_2')$$
$$\xrightarrow{(i_1,K_1,j_1):\overline{b}\langle\nu a_{\Delta+i_2}\rangle} \nu a_{\Delta+i_1+i_2}(X_1' \mid X_2')$$

as desired. The labels of transitions t_1 and t_1' are not equal; they are label equivalent, i.e. $\mu_1 =_\lambda \mu_1'$.
– Rule CAUSE REF applied on t_2. We have

$$\nu a_\Delta(X_1 \mid X_2) \xrightarrow{(i_1,K_1,j_1):\overline{b}\langle\nu a_\Delta\rangle} \nu a_{\Delta+i_1}(X_1' \mid X_2)$$
$$\xrightarrow{(i_2,K_2,j_2):\alpha} \nu a_{\Delta+i_1}(X_1' \mid X_2')$$

with $X_2 \xrightarrow{(i_2,K,j_2):\alpha} X_2'$ and $a \in sub(\alpha)$. If $i_1 \in K_2$ then transition t_1 cause transition t_2 and this is not the case, they are concurrent. If $i_1 \in K$, then by definition of the predicate $\texttt{Cause}(\cdot)$ from the rule CAUSE REF we have three cases: if $\Delta = \Gamma$, then by $i_1 \in K$ and $i_1 \notin K_2$, we have $K \rightsquigarrow_{X_2} K_2$ what implies that i_1 is a synchronisation and that is not the case; if $\Delta = \Gamma_w$, then by the predicate $\texttt{Cause}(\cdot)$, new cause set K_2 is the union of the old cause set K and w, therefore $i_1 \in K$ implies $i_1 \in K_2$, what is not the case; similar if $\Delta = \Gamma_\Omega$. Hence $i_1 \notin K, K_2$ and we can commute transitions:

$$\nu a_\Delta(X_1 \mid X_2) \xrightarrow{(i_2,K_2,j_2):\alpha} \nu a_\Delta(X_1 \mid X_2')$$
$$\xrightarrow{(i_1,K_1,j_1):\overline{b}\langle\nu a_\Delta\rangle} \nu a_{\Delta+i_1}(X_1' \mid X_2')$$

- In the inductive case it is necessary to show that if $X \xrightarrow{\mu_1} Y \xrightarrow{\mu_2} Z$ and $X \xrightarrow{\mu_2'} Y_1 \xrightarrow{\mu_1'} Z$, then the following holds

$$\nu a_\Delta X \xrightarrow{\mu_1} \nu a_{\Delta_1'} Y \xrightarrow{\mu_2} \nu a_{\Delta'} Z \text{ and } \nu a_\Delta X \xrightarrow{\mu_2'} \nu a_{\Delta_2'} Y_1 \xrightarrow{\mu_1'} \nu a_{\Delta'} Z$$

$$Z_i \mid X \xrightarrow{\mu_1} Z_i \mid Y \xrightarrow{\mu_2} Z_i \mid Z \text{ and } Z_i \mid X \xrightarrow{\mu_2'} Z_i \mid Y_1 \xrightarrow{\mu_1'} Z_i \mid Z$$

  Both subcases are straightforward.

2. t_2 is a synchronisation and t_1 is not. We observe the case when t_1 is performing an action $\overline{b}\langle \nu a_\Delta \rangle$ (the rest of the cases are straightforward). In this case, the applied rule could be OPEN or OPEN$^\bullet$.

   If t_2 is a synchronisation which does not involve name a or involve just one component of the process X, then by Property 5 the case is trivial.

   We shall consider the case when transition t_2 involves two contexts $C_i[\bullet]$ and $C_j[\bullet]$ of the process X written as:

$$X = \nu a_{\Delta_n} C_n[\ldots \nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\nu a_{\Delta_0} C_0[X_1]]]]$$

   and name a is used in the object position of the label α_2 of t_2. Then rules CLOSE and CLOSE$^\bullet$ can be applied. Now we proceed with the induction on the structure of the process.

   - In the base case, since transitions will modify contexts just up to νa_{Δ_i}, we can reason on process X written as:

$$X = \nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[X_1]]]$$

     Now we combine rules applied on transition t_1 with the one applied on t_2 and we have:
     – Rule OPEN applied on t_1 and rule CLOSE on t_2. We assume that transition t_1 is executed on the component X_1. Then we have:

$$\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[X_1]] \xrightarrow{(i_1, K_1, j_1):\overline{b}\langle \nu a_{\Delta_i} \rangle} \twoheadrightarrow$$
$$\nu a_{\Delta_i + i_1} C_i[\ldots \nu a_{\Delta_j + i_1} C_j[X_1']] \xrightarrow{(i_2, *, *):\tau} \twoheadrightarrow$$
$$\nu a_{\Delta_i + i_1} \nu a_{\Delta' + i_1} C_i'[\ldots \nu a_{\Delta_j' + i_1} C_j'[X_1']] = Z$$

where $\nu a_{\Delta+i_1} \in C_i[\ldots \nu a_{\Delta_j+i_1} C_j[X_1']]$. To permute transitions, we need to be sure that transition t_1 do not cause transition t_2. Since t_2 is synchronisation, we need to check if t_1 cause transition that are involved in the communication, but then we have to check if lemma hold for transition that are not synchronisations and that is done in the case 1. Hence, we commute transitions and obtain:

$$\begin{aligned}
&\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[X_1]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i} \nu a_{\Delta'} C_i'[\ldots \nu a_{\Delta_j'} C_j'[X_1]] \xrightarrow{(i_1,K_1,j_1):\overline{b}\langle \nu a_{\Delta_i}\rangle}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i+i_1} \nu a_{\Delta'+i_1} C_i'[\ldots \nu a_{\Delta_j'+i_1} C_j'[X_1']] = Z
\end{aligned}$$

where $\nu a_{\Delta} \in C_i[\ldots \nu a_{\Delta_j} C_j[X_1]]$. We have $Z = Z$, as desired.
– Rule OPEN$^\bullet$ applied on t_1 and rule CLOSE on the t_2. We have:

$$\begin{aligned}
&\nu a_{\Delta_i+i_1} C_i[\ldots \nu a_{\Delta_j+i_1} C_j[X_1]] \xrightarrow{(i_1,K_1,j_1):\overline{b}\langle \nu a_{\Delta_i}\rangle} \\
&\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[X_1']] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i} \nu a_{\Delta'} C_i'[\ldots \nu a_{\Delta_j'} C_j'[X_1']] = Z
\end{aligned}$$

where $\nu a_{\Delta} \in C_i[\ldots \nu a_{\Delta_j} C_j[X_1']]$. It is not possible that backward transition t_1 cause transition t_2 and we can swap transitions and obtain:

$$\begin{aligned}
&\nu a_{\Delta_i+i_1} C_i[\ldots \nu a_{\Delta_j+i_1} C_j[X_1]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i+i_1} \nu a_{\Delta'+i_1} C_i'[\ldots \nu a_{\Delta_j'+i_1} C_j'[X_1]] \xrightarrow{(i_1,K_1,j_1):\overline{b}\langle \nu a_{\Delta_i}\rangle} \\
&\nu a_{\Delta_i} \nu a_{\Delta'} C_i'[\ldots \nu a_{\Delta_j'} C_j'[X_1']] = Z
\end{aligned}$$

where $\nu a_{\Delta+i_1} \in C_i[\ldots \nu a_{\Delta_j+i_1} C_j[X_1]]$.
– Similarly for the rules OPEN and OPEN$^\bullet$ combined with rule CLOSE$^\bullet$.

- The inductive case is trivial since t_1 only modifies processes in the context, not the context by itself. Considering the synchronisation $t : X_1 \xrightarrow{(i,*,*):\tau} X_1'$, we have $t : C[\nu a_\Delta (X_1 \mid X_2)] \xrightarrow{(i,*,*):\tau}$

$C[\nu a_\Delta(X_1' \mid X_2)]$ where we can notice that transition t modified just X_1 (Property 5).

3. The case when t_1 is a synchronisation and t_2 is not, is similar to the one above.

4. t_1 and t_2 are synchronisations. We consider the cases when synchronisations involve two different components of the process X. Assume that t_1 is a synchronisation between process X_1 where output is executed and context C_j where input is performed; while t_2 is a synchronisation between input in context C_i and output in C_k. Since transitions will modify contexts just up to νa_{Δ_i}, we can reason on process X written as:

$$X = \nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]]$$

We continue with the case analysis depending whether name a is in the subject or in the object position in the transitions t_1 and t_2.

- name a is in the object position in both transitions; in this case, rules CLOSE and CLOSE$^\bullet$ can be used. Let us consider the case when rule CLOSE is applied on t_1 and on t_2; the rest of the cases are similar. We have

$$\begin{aligned}
&\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]] \xrightarrow{(i_1,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta'} C_j'[\ldots \nu a_{\Delta_k'} C_k[X_1']]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i} \nu a_{\Delta_l''} C_i'[\ldots \nu a_{\Delta_j'} \nu a_{\Delta''} C_j'[\ldots \nu a_{\Delta_k''} C_k'[X_1']]]
\end{aligned}$$

where $\nu a_\Delta \in C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]$ and $\nu a_{\Delta_l'} \in C_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta'} C_j'[\ldots \nu a_{\Delta_k'} C_k[X_1']]]$. To permute transitions, we need to ensure that transition t_1 is not the cause of transition t_2. Since both of transitions are synchronisations we need to check if the output transition involved in t_1 causes the output transition involved in t_2. This is covered with the case 1. when we proved that lemma holds for a single transitions.

Now we can safely swap the transitions and transition t_2' is:

$$\begin{aligned}
&\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow \\
&\nu a_{\Delta_i} \nu a_{\Delta_l''} C_i'[\ldots \nu a_{\Delta_j'} C_j[\ldots \nu a_{\Delta_k'} C_k'[X_1]]]
\end{aligned}$$

where $\nu a_{\Delta_l} \in C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]]$. The derivation for the transition t'_1 is:

$$\nu a_{\Delta_i} \nu a_{\Delta''_l} C'_i[\ldots \nu a_{\Delta'_j} C_j[\ldots \nu a_{\Delta'_k} C'_k[X_1]]] \xrightarrow{(i_1,*,*):\tau}\!\!\!\!\rightarrow$$
$$\nu a_{\Delta_i} \nu a_{\Delta''_l} C'_i[\ldots \nu a_{\Delta'_j} \nu a_{\Delta''} C'_j[\ldots \nu a_{\Delta''_k} C'_k[X'_1]]]$$

where $\nu a_{\Delta'} \in C_j[\ldots \nu a_{\Delta'_k} C'_k[X_1]]$, as desired.

- name a is in the object position in transition t_1 and in the subject in t_2. In this case, rules CLOSE and CLOSE$^\bullet$ can be applied on t_1 and COM and COM$^\bullet$ on t_2. Let us consider the case when CLOSE is applied on t_1 and COM on t_2. The rest of the cases are similar. By executing rule CLOSE on t_1, we have:

$$\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]] \xrightarrow{(i_1,*,*):\tau}\!\!\!\!\rightarrow$$
$$\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta+i_1} C'_j[\ldots \nu a_{\Delta_k+i_1} C_k[X'_1]]]$$

where $\nu a_{\Delta} \in C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]]$. By executing the rule COM we are changing just contexts C_i and C_k, not the restrictions.

$$\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta+i_1} C'_j[\ldots \nu a_{\Delta_k+i_1} C_k[X'_1]]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow$$
$$\nu a_{\Delta_i} C'_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta+i_1} C'_j[\ldots \nu a_{\Delta_k+i_1} C'_k[X'_1]]]$$

To permute transitions, we need to ensure that transition t_1 is not the cause of transition t_2. Since both of transitions are synchronisations we need to check if the output action involved in t_1 causes the output or the input action involved in transition t_2. This is covered with the case 1. when we proved that lemma holds for a single transitions t_1 and t_2. Hence, we can swap transitions and obtain:

$$\nu a_{\Delta_i} C_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C_k[X_1]]] \xrightarrow{(i_2,*,*):\tau}\!\!\!\!\rightarrow$$
$$\nu a_{\Delta_i} C'_i[\ldots \nu a_{\Delta_j} C_j[\ldots \nu a_{\Delta_k} C'_k[X_1]]] \xrightarrow{(i_1,*,*):\tau}\!\!\!\!\rightarrow$$
$$\nu a_{\Delta_i} C'_i[\ldots \nu a_{\Delta_j} \nu a_{\Delta+i_1} C'_j[\ldots \nu a_{\Delta_k+i_1} C'_k[X'_1]]]$$

where $\nu a_{\Delta} \in C_j[\ldots \nu a_{\Delta_k} C'_k[X_1]]]$ since transition t_2 does not change restrictions. $\square$

We use the standard notation and write $t_2 = t_2'/t_1$ for a residual of t_2' after t_1. Additionally, we bring from the π-calculus standard notions on the transitions (Definition 25). Two transitions that have the same source, are called *coinitial*; if they have the same target, they are *cofinal* and if target of one is source of the other transition, they are called *composable*. We denote with $t_1; t_2$ a sequence of pairwise composable transitions that is called *trace*. *Empty* trace is written as ϵ.

Lemma 22. (**Parabolic traces**) Let s be a trace. Then there exist a backward-only trace r and a forward-only trace r' such that $s \sim r; r'$.

Proof The proof is by induction on the length of s and the distance between the very first transition in s and the pair of transitions contradicting the statement of the lemma. Let suppose that this is a pair of transitions $t_1; t_2$. Then we have:

$$t_1 : X \xrightarrow{(i_1,k_1,j_1):\alpha_1} Y \qquad t_2 : Y \rightsquigarrow^{(i_2,k_2,j_2):\alpha_2} Z$$

We have two cases depending if the keys of t_1 and t_2 are the same or not.

- if $i_1 = i_2$, then by the fact that keys are unique in a reversible process, to execute from the process Y the transition t_2 with the key $i_2 = i_1$, transition t_2 needs to be the reverse of the transition t_1 i.e. $t_2 = t_1^\bullet$. Hence, we can eliminate transitions t_1 and t_2 (from Definition 61 we have $t_1; t_1^\bullet \sim \epsilon$) and decrease the length of s.

- if $i_1 \neq i_2$, then we have two possibilities:

  - t_1 and t_2 are concurrent; then we can apply Lemma 20 and swap them. In this way we decrease the distance between the very first transition in s and the pair of transitions contradicting the statement of the lemma.

  - t_1 and t_2 are causally dependent; this case is impossible. They cannot be structural or object dependent since transitions t_1 and t_2 are consecutive and t_2 is the backward one. $\square$

Lemma 23. Let us denote with s_1 and s_2 two coinitial and cofinal traces, where s_2 is forward only. Then there exists a forward-only trace s_1', shorter or equal to s_1, such that $s_1 \sim s_1'$.

Proof The proof is by induction on the length of s_1. If s_1 is forward-only then $s_1' = s_1$. If not, by Lemma 22, we can assume that s_1 is parabolic, and write it as $s_1 = u; t_1; t_2; v$ where $t_1; t_2$ is the only pair of consecutive transitions in the opposite direction; $u; t_1$ is backward-only and $t_2; v$ is forward-only. Since traces s_1 and s_2 are coinitial and cofinal and s_2 is a forward-only, we can notice that the history element which transition t_1 takes out of the history, some transition in $t_2; v$ needs to put back, otherwise, the difference will stay visible (i.e. s_1 and s_2 would not be cofinal). Let us denote with t' the first such transition. To preserve the same target in the end of the traces s_1 and s_2 , we have that t' is exact inverse of the transition t_1 i.e. since t_1 is backward transition, we have $t' = t_1^\bullet$. We can rewrite s_1 as $u; t_1; t_2; v_1; t'; v_2$ where trace $t_2; v_1; t'; v_2$ is forward-only.

We proceed by showing that t_1 is concurrent with all transitions up to t'. Let us suppose opposite, that there exists some transition t'' between t_1 and t' such that t_1 and t'' are causal. Depending on the type of cause we can distinguish two cases:

- if t_1 and t'' are structural causal then we have a contradiction with the hypothesis that t' is the first transition that will put back the history element that t_1 deletes.
- the case when t_1 and t'' are object causal is impossible, since t_1 is a backward transition and t'' is forward.

We can conclude that transition t_1 is concurrent with all transitions between t_1 and t'. By Lemma 20 we can swap t_1 with each transitions up to t' and by Definition 61 we have $s_1 \sim u; t_2; v_1; t_1; t'; v_2$. By the same definition (Definition 61) we have $s_1 \sim u; t_2; v_1; v_2$ (since $t_1^\bullet = t'$, we can erase the transitions because $t_1; t' \sim \epsilon$). In this way, the length of s_1 decreases and we can apply the inductive hypothesis. □

Theorem 10. (**Causal-consistency)** Two traces are coinitial and cofinal if and only if they are equivalent up-to permutation.

Proof Let us denote two traces with s_1 and s_2. If $s_1 \sim s_2$ then from the definition of $\sim$ (Definition 61) we can conclude that they are coinitial and cofinal.

Let us suppose that s_1 and s_2 are coinitial and cofinal. From the Lemma 22 we can suppose that they are parabolic. We shall reason by induction on the lengths of s_1, s_2 and on the depth of the very first disagreement between them. We shall denote it with the pair t_1, t_2. Then we can write the traces s_1 and s_2 as

$$s_1 = u_1; t_1; v_1 \qquad s_2 = u_2; t_2; v_2$$

where $u_1 \sim u_2$. Depending on whether t_1 and t_2 are forward or not, we have the following cases:

- t_1 is forward and t_2 is backward. Since s_1 is parabolic we have that u_1 is backward-only and v_1 is forward-only. From $u_1 \sim u_2$ we have that u_1 and u_2 are coinitial and cofinal, hence the traces $t_1; v_1$ and $t_2; v_2$ are coinitial and cofinal (s_1 and s_2 are cofinal) where $t_1; v_1$ is forward only.

  Now we can apply Lemma 23 on the traces $t_1; v_1$ and $t_2; v_2$ and we have that there exists a trace s_2' (forward-only), shorter or equal to $t_2; v_2$ such that $s_2' \sim t_2; v_2$. If it is equal then t_2 needs to be forward and this is in contradiction with the fact that t_2 is backward. If it is shorter then we proceed by induction with $u_2; s_2'$ shorter.

- t_1 and t_2 are forward. Then $t_1; v_1$ and $t_2; v_2$ are coinitial, cofinal and forward-only. We have two cases depending on whether t_1 and t_2 are concurrent or not.

  - if t_1 and t_2 are concurrent then whatever t_1 puts in the history, v_2 needs to do the same. Let t_1' be the first such transition, then $t_1' \in v_2$ and $t_1' =_p t_1$. Now we can rewrite $t_2; v_2$ as $t_2; v_2'; t_1'; v_2''$ and show that t_1' is concurrent with all transitions in v_2':

* t_1' is the first transition on the same prefix as t_1 (since $t_1; v_1$ and $t_2; v_2$ are coinitial, cofinal and forward-only). Hence, it is not structural causal with any transition in $t_2; v_2'$.
* from $t_1' =_p t_1$, cause sets of both transition are the same and since t_1 is coinitial with $t_2; v_2'$ and $t_2; v_2'$ are forward-only, transition t_1 cannot have as contextual cause any transition from $t_2; v_2'$.

We can conclude that transitions t_1 and t_2 are concurrent and from Lemma 20 we have:

$$t_2; v_2 = t_2; v_2'; t_1'; v_2'' \sim t_1'; t_2; v_2'; v_2''.$$

Since $t_1' =_p t_1$, they are on the exactly the same prefix and they are coinitial, from Lemma 21 we have that $t_1' = t_1$. Without changing the length of s_1 and s_2 we obtain the first disagreement pair later and we can rely on the inductive hypothesis.

– the case when t_1 and t_2 are causally related is impossible since they are both forward, and coinitial.

- The proof is similar if both transitions t_1 and t_2 are backward.

Appendix B

In this Section we give the full proofs from Section 5.6.

Lemma 24. (Structural correspondence) Starting from initial π-calculus process P, where $P = \lambda(A_1) = \varphi(X_1)$, we have:

1. if $A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ is a trace in causal semantics [14], then exists a trace $X_1 \xrightarrow{\mu_1} X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n} X_{n+1}^{K_{Fn}}$ and K_{Fi} in the framework, such that for all i, $\lambda(A_i) = \varphi(X_i^{K_{Fi}})$, $\zeta_i = \gamma(\mu_i)$ and $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$, for $i = 1, ..., n$.

2. if $X_1 \xrightarrow{\mu_1} X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n} X_{n+1}^{K_{Fn}}$ is a trace in the framework, then exists a trace $A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ in causal semantics, where for all i, $\lambda(A_i) = \varphi(X_i^{K_{Fi}})$, $\zeta_i = \gamma(\mu_i)$ and $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$, for $i = 1, ..., n$.

Proof Both directions (*1.* and *2.*) are proved by induction on the length of the computation. Let us consider the direction *1.*

(I) The base case is given by a single transition, and there is no cause; hence, $\mathtt{K}_1 = K_{F1} = \emptyset$. We proceed by induction on the structure of the π-calculus process P and applied rule on the transition t, where $t : A_1 \xrightarrow{\zeta_1} A_2$ and $t' : X_1 \xrightarrow{\mu_1} X_2$ with $P = \lambda(A_1) = \varphi(X_1)$.

- $P = \pi.P'$ where $\pi = \overline{b}a$ or $\pi = b(x)$; Rules that can be applied in Boreale and Sangiorgi's semantics are BS-OUT and BS-IN. We show the case when rule BS-OUT is applied; the other case is similar.

We have $\pi.P' \xrightarrow{i_1:\pi} \{i_1\} :: P' = A_2$ where $\lambda(A_2) = P'$.

In the framework we can execute the corresponding action by applying the rule OUT1 and we have $\pi^*.P' \xrightarrow{(i_1,*,*):\pi} \pi^*[i_1,*].\mathbf{P'} = X_2$ with $\pi^* = \overline{b}^* a^*$ and $\varphi(X_2) = P'$ as desired.

- $P = Q \mid Q'$; Rules that can be applied in Boreale and Sangiorgi's semantics are BS-PAR, BS-COM and BS-CLOSE. We show the case when rule BS-CLOSE is used; the rest of the cases are similar.

  Since rule BS-CLOSE is applied on the process $Q \mid Q'$ one of the parallel components needs to extrude a bound name. Let it be a process $Q = \nu a(Q_1)$. Then we have

  $$\nu a(Q_1) \mid Q' \xrightarrow{\tau} \nu a(Q_2 \mid Q''\{^a/_x\}) = A_2$$

  with the premises $\nu a(Q_1) \xrightarrow{i_1:\overline{b}\langle\nu a\rangle} Q_2$ and $Q' \xrightarrow{i_1:b(x)} Q''$, for some name b. Since τ actions do not impose causes and there is no cause set to merge, we have $\lambda(A_2) = \nu a(Q_2 \mid Q''\{^a/_x\})$.

  In the framework we can execute the corresponding synchronisation by applying the rule CLOSE and we have:

  $$\nu a_{\emptyset_*}(Q_1) \mid Q' \xrightarrow{(i_1,*,*):\tau} \nu a_{\emptyset_*}(\nu a_{\{i_1\}_*}(Y_1) \mid Y''\{^{a^{i_1}}/_x\}) = X_2$$

  with the premises $\nu a_{\emptyset_*}(Q_1) \xrightarrow{(i_1,*,*):\overline{b}\langle\nu a_{\emptyset_*}\rangle} \nu a_{\{i_1\}_{i_1}}(Y_1)$, where $\varphi(Y_1) = Q_2$; and $Q' \xrightarrow{(i_1,*,*):b(x)} Y''$ where $\varphi(Y'') = Q''$. Then we have

  $$\varphi(\nu a_{\emptyset_*}(\nu a_{\{i_1\}_*}(Y_1) \mid Y''\{^{a^{i_1}}/_x\})) = \nu a(Q_2 \mid Q''\{^a/_x\})$$

  as desired.

- $P = \nu a(P')$; Rules that can be applied in Boreale and Sangiorgi's semantics are BS-RES and BS-OPEN, depending if the name a belongs to the executing action or not.. We show the case when rule BS-OPEN is applied; the other case is similar to the one above.

If the rule BS-OPEN is applied on the process $\nu a(P')$, executed action extrudes name a, and we have:

$$\nu a(P') \xrightarrow{i_1:\bar{b}\langle\nu a\rangle} \{i_1\} :: P'' = A_2$$

with the premise $P' \xrightarrow{i_1:\bar{b}a} \{i_1\} :: P''$. By discarding cause set $\{i_1\}$ we have $\lambda(A_2) = P''$.

We can match the same action in the framework and apply rule OPEN on the process $\nu a_{\emptyset_*}(P')$ and obtain:

$$\nu a_{\emptyset_*}(P') \xrightarrow{(i_1,*,*):\bar{b}\langle\nu a_{\emptyset_*}\rangle}\!\!\!\!\rightarrow \nu a_{\{i_1\}_{i_1}}(Y') = X_2$$

with the premise $P' \xrightarrow{(i_1,*,*):\bar{b}a}\!\!\!\!\rightarrow Y'$ where $\varphi(Y') = P''$. By discarding the elements of the history from the process X_2, we have $\varphi(X_2) = P''$ as desired.

(II) In the inductive case we let $s_{BS} : A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ be the trace on causal processes and $s_F : X_1 \xrightarrow{\mu_1}\!\!\!\!\rightarrow X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n}\!\!\!\!\rightarrow X_{n+1}^{K_{Fn}}$ the trace in the framework, where $\lambda(A_1) = \varphi(X_1) = P$; and let us suppose that the inductive hypothesis holds for these two traces. By inductive hypothesis, we have that $\lambda(A_i) = \varphi(X_i^{K_{Fi}}) = P_i$ and in the framework there exist sets K_{Fi}, such that $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$ for all $i = 1, \ldots, n$.

To show the inductive step, let

$$t : s_{BS} \xrightarrow[\mathtt{K}_{n+1}]{\zeta_{n+1}} A_{n+2} \qquad \text{and} \qquad t' : s_F \xrightarrow{\mu_{n+1}}\!\!\!\!\rightarrow X_{n+2}^{K_{Fn+1}}$$

be two corresponding computations. We need to prove two statements: (1) $\mathtt{Rem}(K_{Fn+1}) = \mathtt{K}_{n+1}$ and (2) $\lambda(A_{n+2}) = \varphi(X_{n+2}^{K_{Fn+1}})$.

To prove (1) we should look at the action ζ_n because it is the last action that can influence the cause set $\mathtt{K}_{n+1}$ (cause set $\mathtt{K}_{n+1}$ does not depend on the action ζ_{n+1}). There are two main cases:

- action ζ_n is the direct structural cause of the action ζ_{n+1}. Then we have that action ζ_n is a visible action and $\mathtt{K}_{n+1} = \mathtt{K}_n \cup \{i_n\}$. By inductive hypothesis, we have that there exist $K_{Fn}, \mu_n \in s_F$ such

that $\gamma(\mu_n) = \zeta_n$ and $\texttt{Rem}(K_{Fn}) = \texttt{K}_n$. The action μ_n is identified with a key i_n and it is a visible one; therefore we have $K_{Fn+1} = K_{Fn} \cup \{i_n\}$. The method Rem (Definition 67) does not remove keys of the visible actions and we have $\texttt{Rem}(K_{Fn} \cup \{i_n\}) = \texttt{K}_n \cup \{i_n\} = \texttt{K}_{n+1}$ as desired.

- action ζ_n is not the direct cause of the action ζ_{n+1}; then $\zeta_n = i_n : \tau$ or ζ_n happened on a different component in the parallel composition from the action ζ_{n+1}.

  - If $\zeta_n = i_n : \tau$, then there exist $\texttt{K}_j,\ \texttt{K}_h \in s_{BS}$ such that $\texttt{K}_j$ is a cause set of the input action and $\texttt{K}_h$ is a cause set of the output action which participate in the τ move. Since τ actions merge cause sets, we have that $\texttt{K}_{n+1} = \texttt{K}_j \cup \texttt{K}_h$.

  In the framework, a τ move is composed of the same input and output actions as in the trace on the causal processes. Hence, there exist $K_{Fj},\ K_{Fh} \in s_F$, and by inductive hypothesis $\texttt{Rem}(K_{Fj}) = \texttt{K}_j$ and $\texttt{Rem}(K_{Fh}) = \texttt{K}_h$. Since in the framework the τ-action is identified with the key i_n we have that $K_{Fn+1} = K_{Fj} \cup K_{Fh} \cup \{i_n\}$. By Definition 67, method Rem removes keys belonging to the τ actions, hence, we have $\texttt{Rem}(K_{Fj} \cup K_{Fh} \cup \{i_n\}) = \texttt{K}_j \cup \texttt{K}_h = \texttt{K}_{n+1}$ as desired.

  - If ζ_n happened on a different component in the parallel composition, there exist $\texttt{K}_{h+1}, \zeta_h \in s_{BS}$ where ζ_h is the last action on the same component in the parallel composition as ζ_{n+1}. Then we have that $\texttt{K}_{n+1} = \texttt{K}_{h+1}$, since ζ_h was the last action before ζ_{n+1}.

  By inductive hypothesis, in the framework, there exist $K_{Fh+1}, \mu_h \in s_F$, where $\gamma(\mu_h) = \zeta_h$ and $\texttt{Rem}(K_{Fh+1}) = \texttt{K}_{h+1}$. By the same observation we have $K_{Fh+1} = K_{Fn+1}$ as desired.

We prove the case (2) by induction on the structure of the π-calculus process P, where $\lambda(A_{n+1}) = \varphi(X_{n+1}^{K_{Fn}}) = P_{n+1}$ and the last applied rule on the transition t, where $t : A_{n+1} \xrightarrow[\texttt{K}_{n+1}]{\zeta_{n+1}} A_{n+2}$ and $t' : X_{n+1}^{K_{Fn}} \xrightarrow{\mu_{n+1}} X_{n+2}^{K_{Fn+1}}$.

The reasoning is similar to that for the base case. (Using the fact that $\mathtt{Rem}(K_{Fn+1}) = \mathtt{K}_{n+1}$, we ensure that for an action ζ_{n+1} there exists just one corresponding action μ_{n+1}, such that $\gamma(\mu_{n+1}) = \zeta_{n+1}$.)

Remark 8 The rule BS-CAU inductively allows a causal process $A = \mathtt{K}_B :: P$ to execute if process P can execute, while the executed action brings its cause set $\mathtt{K}_B$. In the framework, it is done with the rules IN2 and OUT2, which inductively allow a reversible process to move, independent of its history.

Direction *2.* is proved by induction on the length of the computation (similarly to *1.*).

(I) The base case is given by a single transition, and we have that there is no cause, hence, $K_{F1} = K_{B1} = \emptyset$. We proceed by induction on the structure of the π-calculus process P and the applied rule on the transition t, where $t : X_1 \xrightarrow{\mu_1}\!\!\!\!\rightarrow X_2$ and $t' : A_1 \xrightarrow{\zeta_1} A_2$ with $P = \lambda(A_1) = \varphi(X_1)$. The proof continues with showing correspondence between the rules that can be applied on processes X_1 and A_1, similarly to the case *1.*

(II) In inductive case we let $s_F : X_1 \xrightarrow{\mu_1}\!\!\!\!\rightarrow X_2^{K_{F1}} \ldots X_n^{K_{Fn-1}} \xrightarrow{\mu_n}\!\!\!\!\rightarrow X_{n+1}^{K_{Fn}}$ be the trace in the framework and $s_{BS} : A_1 \xrightarrow[\mathtt{K}_1]{\zeta_1} A_2 \ldots A_n \xrightarrow[\mathtt{K}_n]{\zeta_n} A_{n+1}$ be the trace on causal processes, where $\varphi(X_1) = \lambda(A_1) = P$. We suppose that the inductive hypothesis holds for these two traces and we have that $\varphi(X_i^{K_{Fi}}) = \lambda(A_i) = P_i$ and $\mathtt{Rem}(K_{Fi}) = \mathtt{K}_i$ for all $i = 1, \ldots, n$.

To prove the inductive step, let

$$t : s_F \xrightarrow{\mu_{n+1}}\!\!\!\!\rightarrow X_{n+2}^{K_{Fn+1}} \qquad \text{and} \qquad t' : s_{BS} \xrightarrow[\mathtt{K}_{n+1}]{\zeta_{n+1}} A_{n+2}$$

be two corresponding computations. We need to prove two statements: (1) $\mathtt{Rem}(K_{Fn+1}) = \mathtt{K}_{n+1}$ and (2) $\lambda(\varphi(X_{n+2}^{K_{Fn+1}}) = A_{n+2})$.

To prove (1) we should look at the action μ_n because it can influence cause set K_{Fn+1} (cause set K_{Fn+1} does not depend on the action μ_{n+1}). There are three cases:

- action μ_n is the direct structural cause of the action μ_{n+1} and it is a visible action; then we have $K_{Fn+1} = K_{Fn} \cup \{i_n\}$.

By inductive hypothesis, we have that there exist $\mathtt{K}_n, \zeta_n \in s_{BS}$ such that $\gamma(\mu_n) = \zeta_n$ and $\mathtt{Rem}(K_{Fn}) = \mathtt{K}_n$. Since μ_n is visible, ζ_n needs to be too and it is identified with the key i_n. We have $\mathtt{K}_{n+1} = \mathtt{K}_n \cup \{i_n\}$ as desired.

- action μ_n is the direct structural cause of the action μ_{n+1} and it is a silent action. In the framework we have $K_{Fn+1} = K_{Fn} \cup \{i_n\}$, since silent action is identified with the key i_n. Cause set K_{Fn} of the τ action contains cause sets of the communicating actions (input and the output ones). Hence, there exist $K_{Fj},\ K_{Fh} \in s_F$ such that $K_{Fn} = K_{Fj} \cup K_{Fh}$.

  By inductive hypothesis, we know that there exist $\mathtt{K}_j,\ \mathtt{K}_h \in s_{BS}$ such that $\mathtt{Rem}(K_{Fj}) = \mathtt{K}_j$ and $\mathtt{Rem}(K_{Fh}) = \mathtt{K}_h$. Since silent actions on causal processes just merge two cause sets, we have $\mathtt{K}_{n+1} = \mathtt{K}_j \cup \mathtt{K}_h$. Method $\mathtt{Rem}$ (Definition 67) removes keys belonging to τ actions, hence we have $\mathtt{Rem}(K_{Fn} \cup \{i_n\}) = \mathtt{Rem}(K_{Fj} \cup K_{Fh}) = \mathtt{K}_j \cup \mathtt{K}_h = \mathtt{K}_{n+1}$ as desired.

- action μ_n is not the direct cause of the action μ_{n+1}; then μ_n happened on a different component in the parallel composition from the action μ_{n+1}. In this case, there exist $K_{Fh+1}, \mu_h \in s_F$ where μ_h is the last action on the same component in the parallel composition as μ_{n+1}. Hence, action μ_h is direct cause of the action μ_{n+1} and we have the same reasoning as in the cases above.

We prove case (2) by induction on the structure of the π-calculus process P_{n+1}, where $\varphi(X_{n+1}^{K_{Fn}}) = \lambda(A_{n+1}) = P_{n_1}$ and last applied rule on the transitions t and t'. The reasoning is similar to the base case. (Using the fact that $\mathtt{Rem}(K_{Fn+1}) = \mathtt{K}_{n+1}$, we ensure that for an action μ_{n+1} there exists just one corresponding action ζ_{n+1}, such that $\gamma(\mu_{n+1}) = \zeta_{n+1}$). $\square$

Remark 9 In the framework, rule CAUSE REF can be applied to update cause set K of the action using an extruded name in the subject position. This rule does not influence the structure of the process X, it just records actions that have extruded bound names.

References

[1] L. Aceto. GSOS and finite labelled transition systems. *Theor. Comput. Sci.*, 131(1):181–195, 1994. 13

[2] J. Anders, S. Shabbir, S. Hilt, and E. Lutz. Landauer's principle in the quantum domain. In *Proceedings Sixth Workshop on Developments in Computational Models: Causality, Computation, and Physics, DCM 2010, Edinburgh, Scotland, 9-10th July 2010.*, pages 13–18, 2010. 2

[3] C. Aubert and I. Cristescu. Contextual equivalences in configuration structures and reversibility. *J. Log. Algebr. Meth. Program.*, 86(1):77–106, 2017. 5, 6, 22

[4] H. B. Axelsen and R. Glück. Reversible representation and manipulation of constructor terms in the heap. In *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings*, pages 96–109, 2013. 2

[5] G. Bacci, V. Danos, and O. Kammar. On the statistical thermodynamics of reversible communicating processes. In *CALCO 2011*, volume 6859 of *LNCS*, pages 1–18. Springer, 2011. 2

[6] J. C. M. Baeten and C. Verhoef. A congruence theorem for structured operational semantics with predicates. In E. Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 477–492. Springer, 1993. 25

[7] K. Barylska, E. Erofeev, M. Koutny, L. Mikulski, and M. Piatkowski. Reversing transitions in bounded petri nets. *Fundam. Inform.*, 157(4):341–357, 2018. 3

[8] K. Barylska, A. Gogolinska, L. Mikulski, A. Philippou, M. Piatkowski, and K. Psara. Reversing computations modelled by coloured petri nets. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis*

of Event Data 2018 Satellite event of the conferences: 39th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2018 and 18th International Conference on Application of Concurrency to System Design ACSD 2018, Bratislava, Slovakia, June 25, 2018., pages 91–111, 2018. 3

[9] K. Barylska, M. Koutny, L. Mikulski, and M. Piatkowski. Reversible computation vs. reversibility in petri nets. *Sci. Comput. Program.*, 151:48–60, 2018. 3

[10] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011. 134

[11] J. A. Bergstra, A. Ponse, and S. A. Smolka. *Handbook of process algebra*. Elsevier, 2001. 3

[12] A. Berut, A. Arakelyan, A. Petrosyan, S. Ciliberto, R. Dillenschneider, and E. Lutz. Experimental verification of Landauer' s principle linking information and thermodynamics. *Nature*, 483(7388):187–189, 03 2012. 2

[13] B. Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 299–310, New York, NY, USA, 2000. ACM. 2

[14] M. Boreale and D. Sangiorgi. A fully abstract semantics for causality in the π-calculus. *Acta Inf.*, 35(5):353–400, 1998. 7, 8, 9, 11, 12, 13, 14, 75, 76, 78, 91, 92, 93, 102, 105, 107, 123, 124, 125, 129, 130, 131, 132, 133, 134, 147

[15] G. Boudol and I. Castellani. Permutation of transitions: An event structure semantics for CCS and SCCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 411–427. Springer, 1988. 121

[16] G. Boudol and I. Castellani. Flow models of distributed computations: Three equivalent semantics for CCS. *Inf. Comput.*, 114(2):247–314, 1994. 134

[17] N. Busi and R. Gorrieri. A petri net semantics for pi-calculus. In *CONCUR Philadelphia, PA, USA, August 21-24, 1995, Proceedings*, pages 145–159, 1995. 8, 9, 91

[18] L. Cardelli and C. Laneve. Reversible structures. In *Computational Methods in Systems Biology, 9th International Conference, CMSB 2011, Paris, France, September 21-23, 2011. Proceedings*, pages 131–140, 2011. 2, 5

[19] S. Crafa, D. Varacca, and N. Yoshida. Event structure semantics of parallel extrusion in the pi-calculus. In *FOSSACS 2012*, volume 7213 of *LNCS*, pages 225–239. Springer, 2012. 9, 12, 76, 80, 91, 102, 105, 110, 111, 131, 132, 133, 134

[20] I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible π-calculus. In *LICS 2013*, pages 388–397, 2013. 5, 10, 12, 76, 85, 88, 92, 94, 95, 105, 106, 110, 112, 114, 133, 134

[21] I. Cristescu, J. Krivine, and D. Varacca. Rigid families for the reversible π-calculus. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, pages 3–19, 2016. 10

[22] I. D. Cristescu, J. Krivine, and D. Varacca. Rigid families for CCS and the π-calculus. In *ICTAC*, volume 9399 of *LNCS*, pages 223–240. Springer, 2015. 5, 6, 110

[23] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK, August 31 - September 3, 2004, Proceedings*, pages 292–307, 2004. 3, 4, 5, 10, 15, 17, 19, 22, 23, 24, 29, 67, 69, 70, 112, 114, 121, 133

[24] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, pages 398–412, 2005. 2, 5

[25] V. Danos and J. Krivine. Formal molecular biology done in CCS-R. *Electr. Notes Theor. Comput. Sci.*, 180(3):31–49, 2007. 2, 5

[26] P. Degano and C. Priami. Non-interleaving semantics for mobile processes. *Theor. Comput. Sci.*, 216(1-2):237–270, 1999. 9, 91

[27] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002. 2

[28] J. Engblom. A review of reverse debugging. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–6, Sept 2012. 2

[29] M. P. Frank. Physical foundations of landauer's principle. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, pages 3–33, 2018. 2

[30] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*, pages 370–384, 2014. 2

[31] D. Gorla. Towards a unified approach to encodability and separation results for process calculi. In *CONCUR 2008, 19th International Conference, Toronto, Canada*, volume 5201 of *Lecture Notes in Computer Science*, pages 492–507. Springer, 2008. 31

[32] G. Gößler, O. Sokolsky, and J.-B. Stefani. Counterfactual causality from first principles? In *CREST@ETAPS 2017, Uppsala, Sweden, 29th April 2017.*, pages 47–53, 2017. 134

[33] J. Grattage. A functional quantum programming language. In *LICS*, pages 249–258, Washington, DC, USA, 2005. IEEE Computer Society. 2

[34] E. Graversen, I. Phillips, and N. Yoshida. Event structure semantics of (controlled) reversible CCS. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, pages 102–122, 2018. 5, 6

[35] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. 2

[36] F. Harary. *Graph theory*. Addison-Wesley, 1991. 125, 127

[37] T. T. Hildebrandt, C. Johansen, and H. Normann. A stable non-interleaving early operational semantics for the pi-calculus. In *Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings*, pages 51–63, 2017. 10, 134

[38] J. Hoey, I. Ulidowski, and S. Yuen. Reversing imperative parallel programs. In *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017.*, pages 51–66, 2017. 2

[39] J. Hoey, I. Ulidowski, and S. Yuen. Reversing parallel programs with blocks and procedures. In *Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS 2018, Beijing, China, September 3, 2018.*, pages 69–86, 2018. 2, 3

[40] L. J. Jagadeesan and R. Jagadeesan. Causality and true concurrency: A data-flow analysis of the pi-calculus (extended abstract). In *Algebraic Methodology and Software Technology, 4th International Conference, AMAST '95, Montreal, Canada, July 3-7, 1995, Proceedings*, pages 277–291, 1995. 8

[41] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009. 3

[42] G. B. L. Jr. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1):50–87, 1986. 2

[43] J. Krivine. A verification technique for reversible process algebra. In *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3, 2012. Revised Papers*, pages 204–217, 2012. 5, 15, 17, 19, 22, 24, 29, 67, 70

[44] S. Kuhn and I. Ulidowski. Towards modelling of local reversibility. In *Reversible Computation - 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*, pages 279–284, 2015. 6

[45] S. Kuhn and I. Ulidowski. A calculus for local reversibility. In *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, pages 20–35, 2016. 5, 6

[46] S. Kuhn and I. Ulidowski. Local reversibility in a calculus of covalent bonding. *Sci. Comput. Program.*, 151:18–47, 2018. 2, 5, 6

[47] O. Laadan and J. Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 25:1–25:14, Berkeley, CA, USA, 2007. USENIX Association. 2

[48] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, 1961. 1

[49] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J. Stefani. Concurrent flexible reversibility. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, pages 370–390, 2013. 2

[50] I. Lanese, D. Medic, and C. A. Mezzina. Static vs dynamic reversibility in ccs. *submitted to Acta Informatica*, 2018. 12, 30, 32

[51] I. Lanese, C. Mezzina, and J.-B. Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016. 11, 93, 94, 112, 130

[52] I. Lanese, C. A. Mezzina, and J. Stefani. Reversing higher-order pi. In P. Gastin and F. Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 478–493. Springer, 2010. 11

[53] I. Lanese, C. A. Mezzina, and F. Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014. 134

[54] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. Cauder: A causal-consistent reversible debugger for Erlang. In J. P. Gallagher and M. 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. 2

[55] J. Lévy. An algebraic interpretation of the *lambda beta* k-calculus; and an application of a labelled *lambda* -calculus. *Theor. Comput. Sci.*, 2(1):97–114, 1976. 121

[56] C. Lutz. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986. 2

[57] D. Medic and C. Mezzina. Towards parametric causal semantics in π-calculus. In *Joint Proceedings of the 18th Italian Conference on Theoretical Computer Science and the 32nd Italian Conference on Computational Logic, Naples, Italy, September 26-28.*, pages 121–125, 2017. 92

[58] D. Medic and C. A. Mezzina. Static VS dynamic reversibility in CCS. In S. J. Devitt and I. Lanese, editors, *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*, volume 9720 of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2016. 12, 30, 32, 49, 66

[59] D. Medic, C. A. Mezzina, I. Phillips, and N. Yoshida. A parametric framework for reversible π-calculi. In *Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS 2018, Beijing, China*, pages 87–103, 2018. 12, 92

[60] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 3, 15, 133

[61] U. Montanari and M. Pistore. Concurrent semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 1:411–429, 1995. 8

[62] C. Palamidessi. Comparing the expressive power of the synchronous and asynchronous pi-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003. 31, 66

[63] R. Perera and J. Cheney. Proof-relevant π-calculus: a constructive account of concurrency and causality. *Mathematical Structures in Computer Science*, pages 1–37, 2017. 134

[64] A. Philippou and K. Psara. Reversible computation in petri nets. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, pages 84–101, 2018. 3

[65] I. Phillips and I. Ulidowski. Reversibility and models for concurrency. *Electr. Notes Theor. Comput. Sci.*, 192(1):93–108, 2007. 5, 6

[66] I. Phillips and I. Ulidowski. Reversibility and asymmetric conflict in event structures. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, pages 303–318, 2013. 3, 6

[67] I. Phillips and I. Ulidowski. Reversibility and asymmetric conflict in event structures. *J. Log. Algebr. Meth. Program.*, 84(6):781–805, 2015. 5, 6

[68] I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation, 4th International Workshop, RC 2012, Copenhagen, Denmark, July 2-3, 2012. Revised Papers*, pages 218–232, 2012. 2, 3, 5

[69] I. Phillips, I. Ulidowski, and S. Yuen. Modelling of bonding with processes and events. In *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings*, pages 141–154, 2013. 2, 3

[70] I. C. C. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2):70–96, 2007. 4, 13, 15, 24, 25, 27, 29, 67, 93, 94, 103, 112, 133

[71] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004. 4, 25

[72] D. Sangiorgi. *Expressing mobility in process algebras : first-order and higher-order paradigms*. PhD thesis, University of Edinburgh, UK, 1993. 94

[73] D. Sangiorgi and D. Walker. *The Pi-Calculus - a Theory of Mobile Processes*. Cambridge Uni. Press, 2001. 3, 72, 133

[74] U. P. Schultz. Reversible object-oriented programming with region-based memory management - work-in-progress report. In *Reversible Computation - 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings*, pages 322–328, 2018. 2

[75] J. J. P. Tsai, K. Fang, H. Chen, and Y. Bi. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *IEEE Trans. Software Eng.*, 16(8):897–916, 1990. 2

[76] I. Ulidowski, I. Phillips, and S. Yuen. Concurrency and reversibility. In *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*, pages 1–14, 2014. 3

[77] I. Ulidowski, I. Phillips, and S. Yuen. Reversing event structures. *New Generation Comput.*, 36(3):281–306, 2018. 5, 6

[78] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, pages 43–54, 2008. 2

[79] T. Yokoyama, H. B. Axelsen, and R. Glück. Towards a reversible functional language. In *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers*, pages 14–29, 2011. 2

[80] T. Yokoyama, H. B. Axelsen, and R. Glück. Minimizing garbage size by generating reversible simulations. In *Third International Conference on Networking and Computing, ICNC 2012, Okinawa, Japan, December 5-7, 2012*, pages 379–387, 2012. 2

[81] M. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566–, Sept. 1973. 2

SOME RIGHTS RESERVED

Unless otherwise expressly stated, all original material of whatever nature created by Doriana Medić and included in this thesis, is licensed under a Creative Commons Attribution Noncommercial Share Alike 2.5 Italy License.

Check creativecommons.org/licenses/by-nc-sa/2.5/it/ for the legal code of the full license.

Ask the author about other uses.