Réversibilité dans le pi calcul d'ordre supérieur

Claudio Antares Mezzina

UNIVERSITÉ DE GRENOBLE

THÈSE préparée dans le cadre d'une cotutelle entre L' UNIVERSITÉ DE GRENOBLE et L'UNIVERSITÁ DI BOLOGNA

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

et

DOTTORE DI RICERCA DELL'UNIVERSITÁ DI BOLOGNA

Spécialité : **Informatique**

Arrêté ministérial : 6 janvier 2005 relatif à la cotutelle internationale de thèse
7 Août 2006 relatif à la formation doctorale

Présentée par

Claudio Antares Mezzina

Thèse dirigée par **Jean-Bernard Stefani**
et codirigée par **Davide Sangiorgi**

préparée au sein du **Project SARDES - équipe INRIA / LIG - Dipartimento di Scienze dell'Informazione**
et de **ED MSTII**

Réversibilité dans le pi calcul d'ordre supérieur

Reversing execution in Higher-Order π

Thèse soutenue publiquement le **07/02/2012**,
devant le jury composé de :

Mr, Rocco De Nicola
Professeur IMT Lucca, Président
Mr, Rocco De Nicola
Professeur IMT Lucca, Rapporteur
Mr, Iain Phillips
Senior Lecturer Imperial College London, Rapporteur
Mr, Jean Krivine
CR2 CNRS-PPS Paris, Examinateur
Mr, Jean-Bernard Stefani
Directeur de Recherche Inria, Directeur de thèse
Mr, Davide Sangiorgi
Professeur Université de Bologne, Co-Directeur de thèse

Résumé

Le concept de réversibilité est ancien, mais il soulève de nos jours beaucoup d'intérêt. Il est en effet exploité dans de nombreux domaines tels que la conception de circuits, le débogage et le test de programmes, la simulation et l'informatique quantique. L'idée d'un modèle de programmation réversible peut se montrer particulièrement intéressante pour la construction de systèmes sûrs de fonctionnement, ne serait-ce que parce que plusieurs techniques connues pour la construction de tels systèmes exploitent une forme ou une autre de retour en arrière ou de reprise. Nous poursuivons dans cette thèse l'étude entreprise avec CCS réversible par Vincent Danos et Jean Krivine, en définissant un pi-calcul d'ordre supérieur réversible ($\rho\pi$). Nous prouvons que le modèle obtenu est causalement cohérent, et que l'on peut encoder fidèlement $\rho\pi$ dans une variante du π-calcul d'ordre supérieur. Nous définissons également une primitive de reprise à grain fin qui permet de contrôler le retour en arrière dans une exécution concurrente. Nous spécifions formellement la sémantique de cette primitive, et nous montrons qu'elle possède de bonnes propriétés, y compris en présence d'opérations de reprise concurrentes. Enfin nous définissons un algorithme concurrent implantant cette primitive de reprise et tous montrons que cet algorithme respecte la sémantique définie.

Mots clés: théorie de la concurrence, calcul de processus, réversibilité, programmation réversible, expressivité de la réversibilité.

Abstract

Reversible computing has a long history. Nowadays, reversible computing is attracting increasing interest because of its potential applications in diverse fields, including hardware design, biological modelling, program debugging and testing and quantum computing. Of particular interest is the application of reversible computation notions to the study of programming abstractions for dependable systems, because several techniques used to build dependable systems rely on some forms of undo or rollback. We continue, in this thesis, the study undertaken on reversible CCS by Vincent Danos and Jean Krivine, by defining a reversible higher-order π-calculus ($\rho\pi$). We prove that reversibility in our calculus is causally consistent and that one can encode faithfully $\rho\pi$ into a variant of HOπ. Moreover we design a fine-grained rollback primitive able to control the rollback of a concurrent execution. We give a formal specification of this primitive and show that it enjoys good properties, even in presence of concurrent conflicting rollbacks. We then devise a concurrent algorithm implementing such a primitive and show that the algorithm respects the defined semantics.

Keywords: concurrency theory, process calculi, reversibility, reversible computing, expressiveness of reversibility.

To Antonella, the love of my life.

Acknowledgement

My deepest gratitude goes to Jean-Bernard Stefani who gave me the chance to do the Ph.D. in France. Having him as supervisor has been truly inspiring; his curiosity and enormous patience allowed me to research into an interesting area such as reversibility. During all these years spent together I learned a lot from him.

I am also in debt with Davide Sangiorgi who put me in contact with the Sardes project and gave me the opportunity to do the Ph.D in "cotutelle" with Italy. Having him first as professor and then as supervisor has been a great honour.

Many thanks to Rocco De Nicola and Iain Phillips who accepted to review my thesis and gave me precious comments on it.

I am also thankful to Jean Krivine for the insightful discussions we had on reversibility. I got really inspired from his works and I believe that without them there would not have been my thesis.

Special thanks goes to Alan Schmitt who gave me the opportunity to join the Sardes team with an intern-ship. Working with him and discussing about process calculi has been really inspiring.

I would like also to thank Ivan Lanese who helped me during these years in continuing the research on reversibility and on proving the hardest proofs of the encoding. I think without him this thesis's results would have been different. Working with him gave me interesting insights on behavioural theory and process calculi in general.

Special thanks goes to Jorge A. Pérez P. who had the patience to read and correct my document. Debating with him about Higher-Order π-calculus is always intriguing. Moreover, I thank him for his friendship and his academic-brotherhood.

I am also thankful to my Italian Inria colleagues and friends: Simone Gasparini, Alessio Pace, Valerio Schiavoni, Daniele Perito and Cinzia Di Giusto. Without them working at Inria would have been more boring.

I would like also to thank my friends in Grenoble who made my stay in France more bearable: Davide Benato and Salvatore Carvelli.

Many thanks go also to Alessandro Mommo, Kreshnik Vukatana, Vivalanaikabahu Somanaikabahu, Luigi Angelé, Livio Nassisi, Marco Romano and Antonio Esposito for being such great friends and having supported me all over these years.

A special thank goes to my brother, Leonardo Gaetano Mezzina, who introduced me into formal methods and whose example always gave me the will to go further.

I would also like to thank Giuseppe Turlione for being such a great friend in those years and for having hosted me several times in Bologna during my visits to the computer science department.

Contents

List of Figures

Chapter 1

Introduction

This thesis studies the formal foundations of a reversible, concurrent computational model. The model is given as a process calculus, and so it captures reversible, concurrent computation by means of a few essential constructs. It can be thus seen as a core language (see [85]) on top of which we can build and study new programming abstractions and their properties. One such abstraction is the communication of messages with complex structure; this is captured in our calculus by allowing the exchange of processes in communications: we thus obtain a higher-order, reversible process calculus. In the rest of this chapter, we comment further on the challenges intrinsic to our approach, and justify further our interest in reversibility and in higher-order concurrency.

Roughly speaking, *reversible* refers to the possibility of undoing any distributed program computation, possibly step-by-step, and to revert it to a state consistent with the past execution. Usually, in a reversible computing model we can distinguish two kinds of evolutions: *forward* (noted as $\twoheadrightarrow$) and *backward* (noted as $\rightsquigarrow$). Forward executions are the usual ones, that can be executed by every non reversible computational model, while the backward ones are proper to a reversible model. Hence, we can say that a reversible model provides automatically a backward step of execution for each forward step that is executed in it. This implies that the model automatically keeps information about forward steps and exploits this information in order to reverse them. Let us suppose that a program executed on a reversible model, moves forward from the state M to the state N, that is $M \twoheadrightarrow N$. Then reversibility means that there exists a backward computation such that from the state N the program can get back to the state M, that is $N \rightsquigarrow M$. More in general this property is valid for any state reached with an unbounded number of execution steps. If from M after a certain number of execution steps we reach a state N_n, and we write $M \twoheadrightarrow^* N_n$ (where $\twoheadrightarrow^*$ indicates several applications of $\twoheadrightarrow$), then we also have that $N_n \rightsquigarrow^* M$. As one may note, a fully reversible computational model is by nature diverging: every single step can be done and undone potentially an unbounded number of times. Taking the above example, by reiterating the execution $M \twoheadrightarrow N \rightsquigarrow M$ indefinitely we obtain a diverging computation.

1.1 Why Reversibility?

The notion of reversible computation has already a long history [12] which started by studies on the thermodynamic cost of irreversible actions. It was noted that since usual computation is irreversible then information loss causes dissipation heat. Therefore it was pointed out that it could be possible to execute reversible computations in a heat dissipation free way. This was the motivation idea that gave rise to several reversible computation models such as *reversible Turing machines* and *conservative logic* [44]. Nowadays, reversible computing is attracting increasing interest because of its potential applications in diverse fields, including hardware design since reversible computations can be performed in a heat dissipation free way; biological modelling since several biological reactions are by nature reversible; program debugging and testing allowing during debugging time to bring the program state back to a certain execution point in which certain conditions are met; and quantum computing. A detailed survey about the history of reversible computing and the different guises it has nowadays can be found in Chapter 2.

Of particular interest is the application of reversible computation notions to the study of programming abstractions for *dependable systems*. Dependable systems are supposed to satisfy a set of distinct properties, such as: safety, security and availability, just to indicate a few of them. Several techniques used to build dependable systems such as *transactions* [48], *system-recovery* schemes [37] and *checkpoint-rollback* protocols [55], rely on some forms of *undo* or *rollback* facility. The ability to undo any single action provides us with an ideal setting to study, revisit, or imagine alternatives to standard techniques for building dependable systems and to debug them. Indeed distributed reversible actions can be seen as *defeasible* partial agreements: the building blocks for different transactional models and recovery techniques. The work of Danos and Krivine, on reversible CCS (RCCS) [29,30], provides a good example: they show how notions of reversible and irreversible actions in a process calculus can model a primitive form of transaction, an abstraction that has been found useful, in different guises, in reliable concurrent and distributed programming, including database [48] and workflow [92] management systems.

1.2 Reversibility in Concurrent Systems

Since process calculi are not confluent and processes are non-deterministic, reversing a computation history means undoing the history not in a deterministic way, but in a causally consistent fashion, where states that are reached during a backward computation are states that could have been reached during the computation history by just performing independent (concurrent) actions in a different order. This is due to the fact that the model does not keep track of the *order*[1] in which concurrent independent actions are executed (scheduled).

In this way, when undoing a computation history we may choose to start with not the *last* executed action, but with some action that has been executed *before* it and that is totally independent from the last one. Therefore it can be undone without respecting the temporal order in which actions have been executed during the forward execution. To better understand this crucial, but simple, concept about reversibility, let us consider the following example:

$$\begin{tikzcd}
& M \arrow[dl, two heads, "a"'] \arrow[dr, two heads, "b"] & \\
M_1 \arrow[dr, two heads, "b"'] & & M_2 \arrow[dl, two heads, "a"] \\
& N &
\end{tikzcd}$$

where from the state M we have two possible executions (paths) to reach the state N. Either we can execute the computation ab (by choosing the left path) or the computation ba (by choosing the right path). Note that the two actions, a and b, are concurrent and independent, since from M both can be executed. That is we can execute either $M \twoheadrightarrow M_1 \twoheadrightarrow N$ or $M \twoheadrightarrow M_2 \twoheadrightarrow N$. Let us suppose that the computation leading to N has been ab. Now from N, it is possible to get back to M by choosing to undo the action a before the action b, reaching so the state M_2 that is a state that we could have reached during the forward computation by just swapping the execution of concurrent actions, in this case executing b before a. Said otherwise, we can

[1] Let us note that in a distributed execution there may not exist a global order among actions.

obtain the following execution $M \twoheadrightarrow M_1 \twoheadrightarrow N \rightsquigarrow M_2 \rightsquigarrow M$.

1.3 Why Higher-Order?

The spread of the Internet and intra-nets has made distributed systems a central area for the development of modern software. Today distributed systems are everywhere: from the world wide web, to the network of digital equipments in a modern car, to our smart-phones and tablets, making even ourselves part of a distributed environment. Moreover with the increasing of bandwidth, messages exchanged in a distributed system are more and more complex in structure, and sometimes they have an autonomous nature. This can be seen in a number of applications of these days [79]:

Plug-ins. Modern browsers, and more generally modern applications, allow the user to download extensions in the form of self-contained programs with the purpose of extending, modifying or removing at run-time functionalities.

Service Oriented Computing. Services are pieces of software proving basic functionalities, which can be accessed, manipulated and composed into complex architectures in order to perform non-trivial tasks.

From the examples above it is clear that code mobility is an increasingly crucial aspect of modern applications. Hence, programming abstractions should account for forms of code mobility in order to be able to support such applications.

In order to use reversibility as an underlying theory for reliable distributed systems, we present a reversible variant of Higher-Order π-calculus (HOπ) [86]. We chose HOπ as our substrate because we find it a convenient starting point for studying distributed programming models with inherently higher-order features such as dynamic code update, which we aim to combine with abstractions for system recovery and fault tolerance. An alternative would have been using the standard (first-order) π-calculus, and then encoding higher-order communication following the well-known representation of higher-order into first-order. In fact, in [86], Sangiorgi provided a fully abstract encoding of HOπ into π, showing that *channel mobility* is enough to encode *code mobility*. Distinction between the two calculi comes when considering explicit notions of localities, useful to model distributed systems. Indeed, as hinted in [89], Sangiorgi's encoding is no longer satisfactory once location-aware primitives (such as locations) are considered. The moral is that while translations such as Sangiorgi's one are satisfactory in the case of *basic* higher-order languages, this is not necessarily the case for higher-order process calculi with specialized constructs [79]. Even if improvements have been made, it would be nice to reverse normal π-calculus and see whether by using the same idea of Sangiorgi's encoding we can obtain a fully abstract encoding of reversible Higher-Order π into reversible π. For all these reasons, we decided to start directly from HOπ building our reversible theory on it. As we will see in the rest of the document the method we will use to reverse HOπ is general enough that it can be applied also to normal π.

1.4 Contributions of the Thesis

The work presented in this thesis addresses three problems: reversing Higher-Order π-calculus [58], controlling reversibility in Higher-Order π-calculus [57] and stating the expressive power of reversibility (partially present in [58]). Let us discuss in detail these contributions.

Reversing Higher-Order π. Reversible CCS [29] is an extension of CCS labelled transition systems (or LTS) (without recursion) in order to support reversibility. In RCCS, each process is monitored by a *memory*, that serves as stack of past actions. Memories are considered as unique process identifiers, and in order to preserve this uniqueness along a parallel composition, a structural law, storing the exact position of each process in a parallel composition, permits to obtain unique memories though a parallel composition. But this rule conflicts with the standard parallel operator properties (*commutativity* and *associativity*). A general method for reversing process calculi has been proposed by Phillips and Ulidowski in [82]. The main idea of this approach is the use of *communication keys* to uniquely identify communications, and to make static each operator (this will be made clear in Section 2.1.2). Unfortunately, this general method is only given for calculi whose operational semantics can be defined using SOS rules conforming to the *path* format, which is not the case for HOπ [77]. In other words, this approach does not apply to calculi using binders or higher-order constructs.

Following the main ideas of these two works, we devise a simple syntax and reduction semantics for a reversible HOπ calculus (for HOπ see [87]), with a novel way of defining reversible reductions by just using simple unique identifiers and memories. This semantics preserves the associativity and commutativity of the parallel operator. Reversible Higher-Order π (also known as $\rho\pi$) is the *first*, to the authors' knowledge, reversible higher order calculus, and we prove that reversibility in $\rho\pi$ is causally consistent, and that $\rho\pi$ is a conservative extension of HOπ.

Controlling Reversibility in Higher-Order π. Rollback recovery has been one of the most widely used means for system recovery in the occurrence of errors. The basic idea behind it is to model the system execution as a succession of system states and, when an error occurs while the system is reaching some state, to roll the system back to a previously reached state and resume execution from that state [90]. Transactions and rollback recovery schemas imply a way to bring the execution back to a certain *specified* previous state. Also reversible debuggers provide primitives to restore the execution of a debugged program through a specific command. For example in [15] the primitive `bstep n` allows to restore the program execution of n steps before the current one. In order to use reversibility as an underlying theory to various techniques for dependability and debugging in distributed systems, we need to control it.

To control reversibility two questions should be answered: *when* backward reductions should be enabled, and *how far back* they apply. The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. We then exploit this intuition to introduce an explicit form of backward reduction. Specifically, backward reduction is not allowed by default as in $\rho\pi$, but has to be triggered by an instruction of the form $\mathsf{roll}\ k$, whose intent is that the current computation be rolled-back to a state just prior to the creation of the memory bearing the tag k. The definition of a proper semantics for such a primitive is a surprisingly delicate matter because of the potential interferences between concurrent rollbacks. We define in this thesis a high-level operational semantics (and calculus, that we call $\mathsf{roll}\text{-}\pi$) which we prove sound and complete with respect to $\rho\pi$ backward reductions. We also define a lower-level distributed semantics using asynchronous notifications and local checks, closer to an actual distributed implementation of the rollback primitive, and we prove it to be fully abstract with respect to the high-level semantics. To the authors' knowledge, this is the first[2] work dealing with controlling reversibility of a reversible calculus.

Expressiveness of Reversibility. We show how it is possible to encode $\rho\pi$ into a variant of HOπ that we call HOπ^+. This variant allows the use of join patterns [42, 43], sub-addressing and abstractions. All these constructs are well known and well understood in terms of expressive power with respect to HOπ (see [72, 88] for functions in π and [43] for join patterns in π). Surprisingly there is no need to use a powerful construct such as passivation (see [64, 91]) in order to mimic the reversible facility of $\rho\pi$. We state the

[2] In some sense the work done in [30] can be considered as controlling reversibility.

faithfulness of our encoding with a finer result than the one presented in [57]. Our new result consists in a weak backward and forward barbed bisimulation (see Section 3.4) between a $\rho\pi$ configuration and its encoding. Such a result, the first in its genre to the authors' knowledge, allows us to encode reversibility in an already existing calculus, without using a specific ad-hoc primitive. Indeed, following the *kernel language* idea [85] (in which all the primitives of a programming language can be built on the top of a restricted set of fundamental primitives, the so called kernel language), if we were to devise a reversible programming language we could choose not to add reversibility as a kernel primitive. It is worth to note that the $\rho\pi$ encoding, along with the results, presented in [57] is different from the one of this thesis. For a better explanation see the introductory part of the Chapter 5. Finally, with a simple modification of the $\rho\pi$ encoding, we are also able to encode roll-π into HOπ^+.

1.5 Outline

The thesis is structured in the following way:

Chapter 2: Reversibility. This chapter provides a deep and wide survey about the state of art of reversibility (and more in general about reversible computing) and abstractions for dependable systems based on system recovery techniques.

Chapter 3: The $\rho\pi$ calculus. This chapter introduces $\rho\pi$, the reversible Higher-Order π-calculus. We first introduce it informally, by showing the main ideas behind our reversing technique, and after that we present it formally. Once the calculus is formally introduced, we show that reversibility in $\rho\pi$ is causally consistent. Naturally we show that $\rho\pi$ is a conservative extension of HOπ.

Chapter 4: The roll-π calculus. In this chapter the roll-π calculus is introduced. It consists just in a extension of $\rho\pi$ with a primitive able to control its reversible facility. As usual we informally introduce this calculus (called roll-π) and we discuss about the problems that such a primitive can cause in presence of concurrent interfering rollbacks. We then provide a high-level semantics for roll-π and show that it is *sound* and *complete* with respect to $\rho\pi$. Finally we give a low-level implementation of such primitive, close to a distributed algorithm, and we show that the implementation respects the original semantics by means of a *full abstraction* result.

Chapter 5: Encodings. In this chapter a variant of HOπ is introduced, called HOπ^+, and then we show that it is possible to encode $\rho\pi$ into HOπ^+. We defend the faithfulness of our encoding by proving that a $\rho\pi$ configuration and its translation behave the same. Finally, we will show how it is possible to encode roll-π into HOπ^+, by just slightly modifying the previous encoding.

Chapter 6: Conclusion and Perspectives. We draw conclusions from our work and discuss perspectives of future work.

Chapters origin. Most of the material present in this thesis has been previously presented in international conferences and appear in their respective proceedings. Even if many improvements have been made to the original papers, we think that the basic idea behind the published results remains the same.

- $\rho\pi$ and its theory presented into Chapter 3 has been published in the paper [58].
- roll-π and its theory presented into Chapter 4 is based on results first published in the paper [57].
- a preliminary version of the $\rho\pi$ encoding and its faithfulness property presented in Chapter 5 has been first published in the paper [58]. The encoding of roll-π is original to this dissertation.

Chapter 2

Reversibility

Thinking about reversibility intuitively leads to thinking also about *undoing*. As hinted by Bennett [12], any forward computation (or execution) can be transformed into a reversible one by just keeping an history of all the information overwritten and hence lost (for example a variable update) by the forward computation, and then use this information to reverse (or undo) the forward computation. Moreover reversibility is also related to *bidirectionality*, which implies a rich literature (see [28]). For the purpose and scope of this document we will just focus on two broad classes of work: *reversible computing*, where the main focuses are to provide a reversible computing model and to see what is the expressive power or the computational cost of reversibility; and *dependable systems abstractions* where dependable systems are obtained by means of some forms of reversibility.

We may classify our work done in Chapter 3 and Chapter 5 ($\rho\pi$) as belonging to the area of reversible computing, and the work done on controlling reversibility (see Chapter 4) as belonging to the area of abstractions for dependable systems. In this way of thinking, in Section 2.1 we will review all the works dealing with finding reversible models and with the expressive power of reversibility. When possible we will compare the reviewed work with respect to ours. On the other hand, in Section 2.2 a considerable variety of works dealing with abstractions for dependable systems will be reviewed and compared with our primitive for controlling reversibility. A few works that we will take into account can be catalogued in both sections, for example the work of Leeman [63] which defines a general way to keep track of past actions in a programming language and provides two interpretations of the *undo* operation. We classify these works, in both sections, by answering the question of what the various authors try to achieve with reversibility.

2.1 Reversible Computing

The first questions raised about reversible computing were about the dissipation of heat (and then loss of energy) that irreversible actions imply. Once that it has been pointed out that reversible actions could be performed in a dissipation-free way, then the following question was to understand if it was possible to simulate reversible actions by irreversible ones, and try to find computational models able to compute reversible computations. These *historical* questions are discussed in Section 2.1.1. In Section 2.1.2 all the reviewed works try to understand what kind of primitives or techniques are necessary in order to reverse both sequential and concurrent programming languages. Section 2.1.3 introduces various works trying to formalize non-standard computational models whose major characteristic is to be inherently reversible. Starting from a reversible model, they try to model it and to understand what are the interesting and useful primitives that can be extrapolated from such models and brought into normal programming languages (somehow the work done in this section is the inverse of Section 2.1.2). Finally Section 2.1.4 considers the question about the link between reversibility and causality.

2.1.1 History

Reversible computing in a broad sense has a long history, which started by studies on the thermodynamic cost of irreversible actions. Indeed Szilard [93] (back in 1925) was already arguing about the thermodynamic cost of information destruction. In 1961, Landauer [56] has demonstrated that it is only the *logically irreversible* operations in a physical computer that necessarily dissipate energy by generating a corresponding amount of entropy for every bit of information that is irreversibly deleted. Hence, logically reversible operations can in principle be performed in a dissipation-free way. Moreover he assumed that at least some logically irreversible operations were necessary to non-trivial computations. In the 70's Bennett started wondering about irreversible actions and he noted that the execution of a reversible program can be distinguished in two halves: the second half was necessary to undo the work done by the first one. The first half would generate the desired answer along with extra information needed by the second half to reverse the computation done by the first one. He then realized [12] that any computation could be made reversible by just accumulating a history of all information that normally would be thrown away and then disposing of this history in order to reverse the process that created it. He then formalized a *reversible Turing machine*. In order to relate his results with thermodynamics and heat dissipation he then started looking for some actual hardware, or some physically reasonable theoretical model able to perform this reversible computation. By seeing an analogy between DNA and RNA and the tapes of a Turing machine he came to hypothesize that an *enzymatic Turing machine* could be implemented. Like Bennett, also Fredkin has studied a computational model able to perform reversible computations, but instead of looking to biology he focused on logic circuits (gates). Indeed he came out with his *Fredkin gate* (and conservative logic [44]): a reversible three inputs three outputs logic function able to simulate all the other logic operations including AND, OR and NOT. In the meanwhile Toffoli [94] proved that *reversible cellular automata* are computationally universal. Moreover in [44] the *billiard-ball* model of computation is given. This model, based on physical effects such as elastic collisions involving balls and fixed reflectors, takes advantage from the fact that a collision between two balls diverts each one from the path it would have followed if the other was absent. Thus a collision can be thought of a two inputs four outputs logic function whose outputs, for input A and B, are respectively[1]:

$$A \wedge B \qquad \neg A \wedge B \qquad A \wedge \neg B \qquad A \wedge B$$

and it is easy to see that from the outputs we can derive again the inputs, that is we can obtain A from $(A \wedge B) \vee (A \wedge \neg B)$ and B from $(A \wedge B) \vee (\neg A \wedge B)$. Intuitively, it is shown that by giving the container a suitable shape (which corresponds to the computer's hardware), and the balls suitable initial conditions (which correspond to the software-program and input data), one can carry out any specified computation. Moreover with the addition of *mirrors* to redirect balls, such collisions can simulate any ordinary logic function. This is the first example of a *ballistic reversible computer*. Margolus [69] showed how to turn Fredkin's model into a simple *reversible 2D cellular automaton* able to perform any arbitrary sequence of Boolean operations. Finally, Benioff [11] and later Feynman [40] showed that it is possible to devise models of reversible computers in quantum mechanics.

When the reversible Turing machine came out, questions about the complexity of simulating irreversible computations by reversible ones were raised ([21,97]). In [21] a general upper bound, on the trade-off between space and time, to simulate irreversible computations by means of reversible ones is given. Moreover for the first time a lower bound on the extra storage space required by reversible computation is given. Lately in [8] it was shown that reversible Turing machines can compute exactly all the injective, computable

[1] where $\wedge$ and $\neg$ are respectively the boolean conjunction and negation.

functions.

2.1.2 Programming Languages

A framework for adding a general *undo* capability to a programming language, is presented in [63]. The author gives a rich survey on the state of art, till the 80's, about the undo operation and all its manifestations in different fields: such as text editors, programming languages, function inversion, backtracking and physics. The paper identifies two interpretations of the undo operation and motivates them by examples. Such interpretations are: $\mathtt{undo_b}$ and $\mathtt{undo_f}$. Let us introduce them by examples showing the basic ideas behind them. Consider that we are editing an empty file by issuing the following four command lines: insert w, insert x, insert y and finally insert z. Then, the history of the effects on the file is the following sequence:

$$\langle \Omega, w, wx, wxy, wxyz \rangle \tag{2.1}$$

where Ω represents the empty file, and $wxyz$ represents the state of the file containing four lines and four characters. Now, let us introduce a time parameter t such that each unit of time consists to one edit command. So, if we apply the function $\mathtt{undo_b}(1)$ to 2.1 we obtain the following history:

$$\langle \Omega, w, wx, wxy \rangle \tag{2.2}$$

if we continue by applying $\mathtt{undo_b}(3)$ to 2.2 we will obtain Ω. That is $\mathtt{undo_b}(\mathtt{t})$ *destroys* the last t actions in the file history. On the other hand, by allowing the undo operation to move forward in the history a second interpretation is possible. For example, if we apply $\mathtt{undo_f}(1)$ to 2.1 we obtain the following history:

$$\langle \Omega, w, wx, wxy, wxyz, wxy \rangle \tag{2.3}$$

and by applying $\mathtt{undo_f}(3)$ to 2.3 we obtain the following history:

$$\langle \Omega, w, wx, wxy, wxyz, wxy, wx \rangle \tag{2.4}$$

So the $\mathtt{undo_f}(t)$ function moves forward by just copying the state of t states before. The main differences between the two operations are: (*i*) $\mathtt{undo_b}$ destroys history while $\mathtt{undo_f}$ just moves forward preserving all the previous states; (*ii*) $\mathtt{undo_b}$ moves the computation history to a point we already have seen before while $\mathtt{undo_f}$ always creates a new history which never existed in the past. To bring these intuitions into programming languages, a notion of *undo-list* is introduced. An undo-list is composed by triplets of the form $\langle a, v, t\rangle$ where a is a variable name, v its value and t the instant in which the value has been assigned to the variable. Hence, the undo-list gives a means for restoring previous values to variables. A computational history is then made of states, each one represented by an undo-list. Two primitives mimicking the semantics of $\mathtt{undo_b}$ and $\mathtt{undo_f}$ are given along with two primitives for undo-list manipulation: one able to increase the instant t of current state variables and the second one able to copy the value of an undo-list into another. The $\mathtt{undo_b}$ interpretation is similar to our rollback primitive given in roll-π (see Chapter 4). Indeed our primitive allows a configuration to get back in history, to a previous state. The slight difference between them is that $\mathtt{undo_b}(t)$ allows us to get back to t previous states, meanwhile our primitive $\mathsf{roll}\ k$ allows us to get back to the state just before the creation of the event k. With a slight modification of our semantics, we can easily implement the primitive $\mathsf{roll}\ t$ that allows us to undo all the communications related to the t-th memory, back in history, which caused the $\mathsf{roll}\ t$ operation. Said otherwise, if we think of the computational history in our model as a tree, then the primitive $\mathsf{roll}\ t$ will rise up the tree of t levels and will cut all the child processes of the reached node. Naturally in a concurrent (distributed) setting it is impossible to state what are the last t steps of the execution. On the other hand, the $\mathtt{undo_f}(t)$ is in contrast with our model, since we require backward steps to be causally consistent. Hence creating a new state which never could have existed before, as $\mathtt{undo_f}$ does, will break this property.

An example of how high-level functional programs can be mapped *compositionally* into a certain kind of *reversible automaton* is given in [4]. Their approach can be seen as a simple compositional compilation from higher-level functional programs into a reversible model of computation. In [32] a compositional translation of the λ-calculus into a form of reversible abstract machine is given. The IAM, or *interaction abstract machine* (a reversible linear head reduction machine) is then introduced and it is shown that the KAM (Krivine abstract machine) and the PAM (pointer abstract machine) of the λ-calculus can be seen as two instances of the IAM.

An innovative way to tackle the problem of compiling a normal function into a reversible computational model, is the one of designing a natively reversible programming language. In this way, when a function f is written into this reversible language, its inverse f^{-1} is given for free: it is just sufficient to execute backward the function f. For example, by just writing the *Fourier transformation* and by changing the direction of execution one can compute its *inverse*. In [98, 99] a reversible sequential programming language is presented. Usually variable updates are irreversible since they imply information loss. The idea behind this language is to use *reversible updates*. In order to do so, the language deals only with integers and all the arithmetic operations are defined as *1st argument injective*. A partial function $\odot : (A \times B) \to C$ is injective in its first argument iff $\forall a, a' \in A$ $\forall b \in B$, we have:

$$a \odot b = a' \odot b \Longrightarrow a = a'$$

Moreover for any operator $\odot$ injective in its first argument there exists an operator $\oslash : (C \times B) \to A$ such that $\forall a \in A$, $\forall b \in B$:

$$((a \odot b) \oslash b) = a$$

We can now define a reversible update. Given a partial function $f : D \to B$ and operator $\odot : (A \times B) \to C$ injective in its first argument, a partial function $g : (A \times D) \to (C \times D)$ is a reversible update with respect to its first argument if it is functionally equivalent to:

$$g(x, y) = (x \odot f(y), y)$$

In this way there is always an inverse for a reversible update:

$$g^{-1}(x, y) = (x \oslash f(y), y)$$

To prove the strength of this new programming language, an implementation of a reversible Turing machine is given. Moreover in [100] a self-interpreter of the language is given and in [9] an abstract reversible machine and its assembly are given. Naturally to be fully reversible a hard assumption is made: the language does not allow I/O operations. This mechanism of making updates reversible is similar to the one used in $\rho\pi$ in the way of saving information. Indeed, in $\rho\pi$ communications are made reversible by means of memories. Memories store the pair $\langle$*input process*, *output process*$\rangle$ that give rise to a communication. Moreover, a reversible update can be easily encoded in $\rho\pi$ by means of communication: it is sufficient to consider a variable as a message on a particular channel containing its actual value. Then, an assignment will just consume (read) the old value from this particular channel and emit, on it, a message containing the new value. Since communications are reversible, then also the encoded update will be reversible. A drawback of this programming language is that, even if it is a full-fledged programming language, it is devised to work only in a sequential setting, meanwhile $\rho\pi$ is thought to deal with concurrent (possibly distributed) programs.

Notions of reversible computation appear also in works on *reversible* *debuggers* [6,15,39] and on computer simulation tools [24]. In [39] a reversible debugger is presented. Its main novelty is that it allows to reverse the state of a (single-thread) program to a point in which a certain condition is satisfied. Usually conditions are tests on variable values. If the variable during the program execution changes its value *monotonically* then it is possible to uniquely identify the closest checkpoint before this condition is satisfied. In order to allow reversibility, the proposed system uses an automatic checkpointing algorithm that mimics virtual memory algorithms for paging. A checkpoint contains information about the memory state and values of the processor registers (e.g. program counter, stack pointer). When the computation has to be reversed till a point in which a condition is satisfied, the closest checkpoint to reach this state is restored and then the instructions are re-executed (interpreted) forward until the condition is met. Naturally reversing I/O operations is crucial and the system allows programmers to create ad-hoc routines to reverse this kind of operations. A different approach is used in [15] where it is given a reversible debugger able to reverse the computation of n steps without using checkpointing mechanisms. Actually this approach cannot be properly considered as a form of reversible computation, since instead of undoing the last n execution steps, the program is re-executed till a point equivalent to going back of n steps. To do so, special counters in charge of counting the number of the steps are added at runtime. In order to exactly re-execute the program and then its I/O operations, all the calls to I/O system primitives are stored. For example if the program reads from the console an input, then this input is saved and when the debugger re-executes the program the same input is used. Our rollback primitive (see Chapter 4) is more related to the work presented in [39] even if it is more precise since it allows to pinpoint the state in which the program has to get back, without the need to re-execute anything. Our approach can easily mimic the reversing facilities of [15] by just undoing the last n steps that caused the rollback operation (see the above discussion on $\mathtt{undo}_{\mathtt{b}}$). But this will require a modification to the semantics of our primitive. Works on reversible debuggers deal just with a sequential setting, and little is said about how to reverse a concurrent multi-processes program. Our calculus natively supports concurrent programs and indeed our rollback primitive allows to reverse a concurrent (possibly distributed) program.

So far all the presented works deal with sequential settings. The first reversible calculus was introduced in [29] (RCCS). It is a fully reversible variant of CCS [70] (without recursion), where each process is endowed with a memory. For example the *monitored* process $m \triangleright P$ represents a RCCS process P monitored by the memory m. Every action performed by a process is then stored in its memory, that serves as *stack* of past actions. During a synchronization, between two processes, every process memory stores the memory of the counterpart process. In this way both memories are bound and a process cannot undo a synchronization unless its counter part is willing too. To better understand how past actions are saved into memories and how they are used to reverse a process, let us see what the main rules of RCCS are:

$$\text{(ACT)}\ m \triangleright \alpha.P + Q \xrightarrow{m:\alpha} \langle *, \alpha, Q\rangle \cdot m \triangleright P$$
$$\text{(ACT}^*\text{)}\ \langle *, \alpha, Q\rangle \cdot m \triangleright P \xrightarrow{m:\alpha_*} \alpha.P + Q$$

From the above two rules (one the inverse of the other) it is simple to grasp the idea behind RCCS. When a process is able to perform an action (prefix) then this action along with the process context is put on the top of the memory. A process is then able to undo its last action by popping out from the memory the needed information. The asterisk "$*$" stored in the memory is a place-holder that possibly will be substituted by the memory belonging to another process with whom the current process is performing a synchronization. This approach, even if simple and immediate, fails when applied to higher-order calculi since the stored information is just enough to reverse prefix operators, and not, let us say, binders that imply a substitution if executed. In $\rho\pi$ we borrow the idea of using memories but instead of using a stack structure we use a flat one, as we will see in Chapter 3. Moreover we store the entire process before each action, in order to fully restore it back.

A more general approach on how to reverse process calculi defined in a particular kind of GSOS format is given in [82], where as example the *CCSk* calculus is presented. With this technique the structure of a process is never destroyed (during the execution) and the progress is noted by marking, with a special identifier, the actions that have been performed. In order to tag the communicating processes, a unique identifier is generated *on-the-fly* during the communication. The operators of a language like CCS can be divided into the *static* operators, where the operator remains present after a transition, and the dynamic operators, where the operator is destroyed by the transition. Dynamic operators are more *forgetful* than static operators. An example of dynamic operator for CCS is the choice "+" operator, while a static one is the parallel "|" operator. Let us recall their (left) rules:

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q}$$

We can note that after the transition the part "+ Q" of the process is forgotten, while "$\mid Q$" not. Since the structure of the process has not to be destroyed, rules to transform dynamic operators into static ones are given. To prove the power of this set of rules, a reversible CCS (called CCSk) is obtained by applying them on CCS. Each time two processes synchronize a new communication key is created, for example:

$$a.P \mid \overline{a}.Q \rightarrow a[m].P \mid \overline{a}[m].Q$$

As one can see from the above example, the two prefixes a and $\overline{a}$ are not destroyed (or forgotten) as in CCS: the execution *passes* just after them. Each time a synchronization happens, a new communication key is generated (m in the example) and the two prefixes participating to the action are marked with it. The resulting process acts like $P \mid Q$ (if doing forward executions) but also keeps information about the previous synchronization, in order to reverse it. This approach even if powerful, relies on global predicates in order to generate new keys and to state which part of a process is allowed to execute, that is which part of a process represents the exact present and which one just represents the past of the current process. Moreover these general rules are not enough to revert calculi with binders and higher-order aspects. From this approach we will just borrow the idea of using unique identifiers to identify communications.

2.1.3 Models of Biochemical process

Foundational studies of reversible and concurrent computations have been largely inspired by areas such as chemical and biological systems, where operations are reversible and only an injection of energy and/or a change of entropy can move computational system in a desired direction.

In [31] a reversible variant of CCS to model biological systems is given. Two properties of the language are provided: *soundness* (reversing computations do not give access to formerly unreachable states) and *expressiveness* (the memorizing schema does not induce fake causal dependencies on backward sequences of actions). The language is then adapted (by adding *multi-actions*) to model complex biological situations such as the transcription of a protein when controlled by a competition between different reactants.

In a *massive concurrent system* (see [22]), unlike concurrent systems, different processes of the same kind are indistinguishable. Their actions can cause effects, but not to the point of being able to identify the precise molecule that caused an effect. So, standard notions of causality and independence of events need to be adapted. A subset of the DSD language[2] (see [81]), called *reversible structures*, able to model such massive concurrent systems is given in [23]. In this setting processes are called *gates*, and are expressed as a sequence of inputs followed by a sequence of outputs. Moreover, the special symbol "ˆ" acts like a *program counter*: it indicates the next gate action. Said otherwise, the cursor ˆ divides a gate into two parts: past actions and future actions. Each time a gate performs a forward (backward) action its cursor ˆ is moved forward (backward) of one position. For example the gate:

$$\hat{}\,a.b.w : \overline{c}$$

represents a gate able to read a signal on a, then a signal on b and then it is able to emit the signal $w : \overline{c}$. Since ˆ points to the beginning of the gate, this implies that the above gate has no history. Identifiers are used to label signals, but they are not unique. When a gate reads a signal, it stores in its structure the identifier of the signal, for example the gate:

$$u : a.v : b.\hat{}\,w : \overline{c}$$

represents a gate that has consumed two signals, $u : \overline{a}$ and $v : \overline{b}$, and is able to emit the signal $w : \overline{c}$. A signal can be released from a gate by just moving on the cursor ˆ and by releasing the labelled output in parallel with the gate. Following the example before we have:

$$u : a.v : b.\hat{}\,w : \overline{c} \rightarrow u : a.v : b.w : \overline{c}\hat{} \mid w : \overline{c}$$

where we can note how the signal $w : \overline{c}$ is emitted by the gate, and how the gate keeps track of this release by simply moving forward the cursor. Naturally this execution can be reverted by getting back the signal and moving back the cursor. That is:

$$u : a.v : b.w : \overline{c}\hat{} \mid w : \overline{c} \rightarrow u : a.v : b.\hat{}\,w : \overline{c}$$

Since signal identifiers are not unique[3], while reversing a signal emission, it may happen that a gate captures back a signal of the same kind of the one it emitted, but not the one it originally generated. This is possible since two signal on the same channel with the same identifier are indistinguishable. A gate of the form:

$$\hat{}\,b.c$$

[2] A language for designing DNA circuits.

[3] This is a property of massive concurrent systems.

represents a gate able to consume a signal on b and then on c. When a signal is consumed, its identifier is stored into the consuming gate and the cursor moves on one step. Following the example before we have:

$$\hat{}b.c \mid u : \overline{b} \to u : b.\hat{}c$$

This execution can be easily reverted by just releasing the signal with its own identifier and moving back the cursor. That is:

$$u : b.\hat{}c \to \hat{}b.c \mid u : \overline{b}$$

In order to relate reversible structures with other existing reversible calculi, multiplicities are dropped down with a syntactic characterization called *coherent* structures. In this way, it is proved that coherent reversible structures can implement the *asynchronous* version of RCCS. Differences between coherent reversible structures and non-coherent ones, are highlighted by a reachability result.

The idea of using a special character to indicate what is the current state of a process and what is its history is simple and intuitive. It somehow recalls the technique used by Phillips and Ulidowski in [82] where the cursor $\hat{}$ is substituted by communication keys since they need to keep causal information about communications. Moreover this technique, even if simple and immediate, fails when applied to higher-order calculi since the stored information is just enough to reverse a signal emission or a signal reading. The relevant aspect of this work it that by compiling the asynchronous variant of RCCS into reversible structures, it is possible to bring RCCS into a real reversible computational model (such as DNA circuits) and then to execute it. It would be nice to see whether $\rho\pi$ can be compiled into DNA circuits or some other reversible computational model, or like [31] see whether $\rho\pi$ is able to model some biochemical reactions or it requires to be extended with ad-hoc primitives.

2.1.4 Causality

The notion of reversibility is strictly related to that of *causality*. Indeed in order to reverse a system, some causality information should be exploited to state for example which events have been caused by another one. Causal consistency is an expected property of a concurrent reversible system. It implies that states reached during backward computations are states that could have been reached during the forward computation by just swapping the order of execution of concurrent independent actions. For example, if we take the following CCS process:

$$a \mid b$$

a possible execution can be ab, that is the process performs the action a and then continues by doing the b. Since the two actions are concurrent (and independent), by undoing the above computation we may choose to undo the action a and we reach a state in which just the action b is done, that is we reach a state that we could have reached in the forward computation by just executing b before a. As we will see in Chapter 4, the way in which process identifiers in $\rho\pi$ are created induces a partial causal order on them. It is possible then to state if a process caused another one. This partial order will be exploited in order to capture all the processes that have been caused by a particular communication. Therefore, reversing a communication is simple: it is sufficient to eliminate all the processes caused by it and to restore back all the processes not directly related (caused) to the communication we want to reverse. This aspect will be made clearer later on in this document.

There is a rich literature about causality in concurrent systems, in particular about process calculi (see [16, 25, 27, 96]). For example, in π-calculus two kinds of causal dependencies are distinguished (see [16, 36]). The first ones are generated by the nesting of prefixes and are called *structural* (or subject) dependencies. For example, if we take the following π process (and execution):

$$\overline{a}.\overline{b} \mid b.\overline{c} \mid a \mid c \rightarrow \overline{b} \mid b.\overline{c} \mid c \rightarrow c \mid \overline{c} \rightarrow \mathbf{0}$$

we note that the synchronization on channel b (second step) causally depends on the first step since the process $\overline{b}$ is guarded by a. Also synchronization on c (third step) depends on the second step and then transitively on the first one. The other kind of dependencies called *link* (or object) dependencies are generated from the binding mechanisms on names. For example if we take the π process:

$$\nu x.\, (\overline{y}\langle x\rangle \mid \overline{x})$$

in a labelled semantics we have that $\overline{x}$ is causally dependent on the scope extrusion of x that is performed by the output $\overline{y}$, that is x is causally dependent on y. Let us note that the first kind of dependencies are present in CCS-like calculi, meanwhile the second ones in calculi allowing channel mobility. The way in which causality is captured in $\rho\pi$ with tags is similar to the one proposed by Boreale and Sangiorgi (see [16]), where each time a synchronization is done causal information belonging to the two synchronizing processes are exchanged. But this will require further investigation to see whether the causal structure captured by $\rho\pi$ is similar or related to the one that emerges from Boreale and Sangiorgi's work. Moreover, our encoding of $\rho\pi$ into a variant of HOπ (given in Chapter 5) is consistent with, though not reducible to, Boreale and Sangiorgi's encoding of π-calculus with causality into π-calculus itself.

Lately Nestmann et al. [80] raised an interesting question about causality in encoding synchrony via asynchrony. Indeed they argue that a good encoding notion (see [47]) should also reflect causality notions. That is, causal dependencies between actions should be preserved by the encoding. This is in line with our work on the encoding of $\rho\pi$ into HOπ^+, even if we base our notion of good encoding on a behavioral equivalence.

2.2 Dependable System Abstractions

A *failure* of a system occurs when its behavior differs from the one that has been specified. The part of the system state leading to the failure is called the *error*. An error is always caused by a *fault*. The fault is the original cause of an error. Hence faults are the cause of errors that may lead to failures [95].

Fault tolerance is a means for achieving dependability despite the likelihood that a system still contains faults and aiming to provide the required service in spite of them [7]. The ability to undo any single action provides us with an ideal setting to study, revisit, or imagine alternatives to standard techniques for building dependable systems and to debug them. Indeed distributed reversible actions can be seen as *defeasible* partial agreements: the building blocks for different transactional models and recovery techniques.

Two techniques are used to achieve fault tolerance: *fault masking*, aimed at removing errors from the system state before failures happen; and *fault treatment* (or handling) which is devoted to prevent faults from being activated again. In this section we will discuss the second kind of techniques. Moreover, recovery from errors in fault-tolerant systems can be characterised into two techniques: *backward* error recovery and *forward* error recovery. Backward error recovery implies somehow the ability of the system to get back to a state before the error happened, and both transactions (see Section 2.2.1) and checkpoint-recovery techniques (see Section 2.2.3) can be classified as backward recovery technique. Our primitive for controlling reversibility can be seen as a primitive that can be useful for supporting backward recovery schemas. On the other hand, forward error recovery involves correcting the error without resorting to reversing previous operations, since in a distributed setting reverting a global state may be unaffordable in terms of resources spent and complexity of the algorithm. In this way exception handling (e.g. [53]) can be classified as a forward recovery technique. Depending on how compensations (see Section 2.2.2) are programmed, they can be considered as forward or backward recovery techniques. Indeed a compensation can be as simple as undoing an original action and then it can be seen as a backward recovery technique or it can be a more complex program.

2.2.1 Transactions

In the classical transactional model [13, 78] transactions are seen as sequences of read and write operations that map consistent database states to consistent states when executed in isolation. A concurrent execution of a set of transactions is represented as an interleaved sequence of read and write operations, and it is said to be serializable if it is equivalent to a serial (non-concurrent) execution. A transaction is a sequence of actions that have to be executed atomically: either it successfully completes (*commits*) and all its effect are visible to the other transactions; otherwise it *fails* and its effects are not visible.

Transactions, originally introduced in the field of DBMS (Data-Base Management Systems) models, provide good concurrency abstraction models in programming languages, since they ensure nice properties, such as atomicity and isolation, difficult to obtain if manually programmed. Indeed, if a developer were to ensure such properties, he would have to design the program relying on low level concurrent programming primitives (e.g. critical sections, semaphores, monitors), which is typically a difficult task, and even more, hard to debug. In this section we will review works dealing with transactional programming languages guaranteeing *atomicity*, *consistency*, *isolation* and *durability* (or ACID) properties. Works modelling non-classical notions of transactions (such as long-running transactions, open-nested transactions) will be reviewed in Section 2.2.2.

In [50] primitives to build composable transaction abstractions in ML are given. Transactions are factored into four separable features: *persistence*, *undoability*, *locking* and *threads* and then each composition of these properties gives rise to different transactional models. Following the idea of adding transactions to Objective Caml, the language AtomCaml [84] has been proposed. The language is endowed with the new function `atom`(f), of type (`unit` $\rightarrow$ `'a`) $\rightarrow$ `'a`, whose purpose is to execute atomically the function f. The basic idea behind this function (and primitive) is that it tries to execute sequentially (hence not interleaved) the entire function block, if during the execution the thread executing it has been pre-empted by the scheduler then the function rolls-back and re-executes again. This implies that the function is executing in a mono-processor setting, where true concurrency does not exist. Moreover there are a few limitations about the side effects that the atomic block can have: input operations are not allowed (since there is no way to reverse them); output operations are always buffered and *flushed* only if the block successfully completes, and exceptions that escape from the atomic block boundaries force the atomic block to complete. Changes made by an atomic block to mutable variables are logged, that is variables are like stacks of values, and in case of failure these changes are reverted. Other works have explored transactional facilities on real distributed programming languages (see [38, 65, 66]).

Different works have approached the design of transactional languages from a semantic point of view, showing good properties. Transactional Featherweight Java [54] is an object calculus with support for nested and multi-threaded transactions. As usual a transaction is delimited by a special block, in this case the `onacid` statement. Each time an `onacid` is executed a new transition (identified by a label) is created, and all the threads executing in it are bound with the transaction label. Each thread is executed into a transactional environments that keeps track of all the read and write operations that the thread performs on objects. Then by varying the semantics of operations on transactional environment two kinds of semantics are given: *versioning* semantics and *strict two phase locking* semantics. The first one mimics the STM (we will discuss about STM later on) logging mechanism, since when a thread enters a new transaction, an empty log corresponding to the transaction is created. This log keeps track of all the objects that are modified within a transaction, for a write operation for example the old value is written in the log. A transaction successfully commits if its log is consistent with the father's one, otherwise it fails. A log is consistent with the father ones, if all the values of the objects read by the child thread have remained unchanged until the committing time. In the strict two phase locking semantics there is no more need of a log mechanism, since before modifying an object, a transaction has to require a lock on it. All the collected locks will be released when the transaction commits. Nested transactions inherit father locks. By using the lock mechanism, there is no way for a transaction to fail.

In [30] the authors show how it is possible to obtain a general notion of transactions by simply introducing *irreversible* actions into RCCS [29]. Irreversible actions may be seen as commits, indeed if two communicating processes synchronize on an irreversible action, this means that they will not get back to a state prior to the irreversible action. In RCCS an irreversible action is a normal action underlined, whose execution causes the freezing of its memory, according to the following rule:

$$(\underline{\text{ACT}})\ m \triangleright \underline{\alpha}.P + Q \xrightarrow{m:\alpha} \langle \circ \rangle \cdot m \triangleright P$$

While executing an irreversible action no useful information about the previous state is stored in the memory. Indeed on the top of the memory is put the symbol $\langle \circ \rangle$ that will act as a *dam*: it is impossible to get back to a state previous of its creation. This operation has side effects on all the processes that communicated with the process who performed an irreversible action, since it is no longer possible to undo previous synchronizations. Encoding a transaction via irreversible actions is simple: a trace of reversible actions ending with an irreversible action in which all the actions have caused the last one, can be considered as executed atomically. That is, if it commits then the irreversible action is executed and since all the previous actions caused it, this implies they that cannot be reverted. If the irreversible action is not executed, then there is a way to undo all the executed actions. Naturally in this way it is only possible to program flat transactions, but these transactions are enough to program classical concurrency problems such as dining philosophers and leader election.

Distributed reversible actions can be seen as defeasible partial agreements: the building blocks for different transactional models and recovery techniques. The work of Danos and Krivine on reversible CCS (RCCS) [30] provides a good example: they show how notions of reversible and irreversible actions in a process calculus can model a primitive form of transaction, an abstraction that has been found useful, in different guises, in reliable concurrent and distributed programming. Their result is general enough, that it could be applied to $\rho\pi$, in order to obtain a general form of transaction in it. Moreover we could see whether all the primitives presented in the above works could be implemented intro $\rho\pi$, using them as a test-bed for $\rho\pi$.

An elegant way to implement atomic transactions, as a concurrency abstraction model, in programming language is the use of STM [52]: *software transactional memory*. Different works [2, 3, 75] have explored the semantics of various STM schemas starting from *strong atomicity*: where a transaction is executed sequentially (and then perfectly isolated), to *weak atomicity* schemas where interleaving between *non-conflicting* transactions is allowed. STM schemas imply the existence of a *global store* (sometimes also referred as global heap).

In strong atomicity schemas there is no need to log all the modifications done to the store by the transaction, since it is executed sequentially and it always commits. In weak cases, each new transaction is endowed with an empty log, on which are stored all the old values of (global store) variables modified by the transaction. A transaction rollback can be done in two different ways: either atomically, that is all the values of the log are reflected at once into the global store, or into multiple steps (and this really mimics undoing all the actions done by the transaction step by step). From the point of view of process calculi and behavioural theory, an STM model for the asynchronous CCS [71] is introduced in [5]. The key idea of the calculus, called ATCCS, is that a transaction is seen as a sequence of read and write operations (parallelism is not allowed into a transaction) and all these operations are written into the transaction log. When a transaction finishes it can be committed or aborted depending on the global environment. If there are enough messages corresponding to the transaction read operations, then at once these message are removed and all the transaction output are released. If it is not the case, the transaction is re-executed with a new empty log. The authors show that this model is powerful enough to express (encode) several concurrency problems and operators, such as: the leader election algorithm, the guarded choice operator and join patterns. Moreover a weak equivalence on atomic expressions (transaction) is given and it is proved to be a *congruence*.

Although our reversible model adopts a message-passing programming style[4] to facilitate faults isolation, the transactional models for STM proposed in these various works can provide useful benchmarks for it. Indeed by using the same arguments of reversible updates done in Section 2.1.2, we could see if the different schemas for STM works be implemented in $\rho\pi$.

2.2.2 Compensations

Our reversible model assumes that every single distributed action can be undone at any time. This property is quite *strict* when dealing with real distributed systems. Indeed, in a distributed system, composed by several parties, it is unthinkable to enforce for example atomicity by using global

[4] There is no use of shared variables and global store.

locks. *Long-running* transactions [45,46], also knows as web-transactions [67], are transactions in which *isolation* and *atomicity* requirements are relaxed. In long-running transactions, instead of assuming perfect rollback upon a failure, they provide support for explicit programming of *compensation* activities. Although a compensation can be as simple as undoing an original action, and this is perfectly matched by our work, it should not be considered as the simply undoing of the original action. Indeed there are actions that by nature cannot be undone (namely *irreversible* actions), for example if we think about printing a file, it is impossible to undo this operation once it has been executed. Several works have tackled the issue of formalizing long-running transactions into well established process calculi such as $Join$-calculus [42] and π-calculus. All these works recognize that a long running transaction is composed by three elements: a *transactional scope* (sometimes with a specific name) executing the transaction, the transaction *body* representing the process that the transaction should execute if everything goes fine, and the transaction *compensation*, representing a *blocked* process that will be executed just in case of failure. All the works we are going to present differ from different aspects such as the way a failure is launched, the way a compensation is composed, or the way nesting is expressed.

$cJoin$ [18,19] extends $Join$-calculus with primitives to program dynamic nested communications. Negotiations (transactions) are processes that execute in a controlled environment until completion, when they commit and make their results observable to the rest of the system. Additionally, they can be explicitly aborted, in which case, suitable compensation programs can be activated to resume a locally consistent state. A process P is bound with its compensation Q. When multiple processes join a transaction, the transactional scope gets enlarged in order to embrace them. Moreover, when a process joins a transaction its compensation is *stored* in parallel with the transaction one. In this way, as the transactional scope enlarges its compensation changes. If a transaction aborts then its compensation is executed, otherwise if it commits then its compensation is discarded.

πt-calculus [14] is an extension of asynchronous π-calculus with long running transactions. A long running transaction is a process of the form:

$$t(P, F, B, C)$$

where P represents the body to be executed in a transactional way; F and B are the *failure* manager and the *failure* bag to be executed if a failure occurs; and C is the *compensation*, to be executed in case a father transaction fails. The bag B is used to store all the compensations of all the child transactions that have committed while the father transaction is still executing. In this way, if the father transaction fails, its failure manager is launched *after* the compensations of all its committed children, using the following rule[5]:

$$t(\texttt{abort}, F, B, C) \to B; F$$

There is no way for a transaction to abort enclosing or sibling transactions, it can just abort itself. The authors show that πt-calculus can be encoded into π-calculus, and they prove its correctness by means of weak barbed bisimulation.

An interesting work considering timing issue in long running transactions has been carried out in [61], where $\texttt{web}\pi$ is introduced. $\texttt{web}\pi$ is an extension of π-calculus with the concept of long running transaction with time. A transaction has the form:

$$\langle P \ ; \ Q \rangle_x^n$$

where P represents the ongoing transaction x, that is the normal execution of the transaction, Q represents the compensation of the transaction x, and n represents the *time* in which the transaction has to execute. Such transaction can fail because of different events: it may receive a failure signal $\overline{x}$ that may rise from the inner process P or from the enclosing environment; or it may fail because the time elapsed. If a transition fails, its compensation is enabled. A committed transaction has the following form:

$$\langle 0 \ ; \ Q \rangle_x^n$$

and it is structurally equivalent to $\mathbf{0}$, that is a committed transaction cannot fail and cannot be compensated. Nested transactions are flattened via structural equivalence. Notwithstanding this flattening, parent transactions may still affect children transactions by means of transactional names. But since a committed transaction cannot be compensated, this means that this model can only express *open nested* transactions [76] and for example *nested failure* model cannot be easily encoded. Moreover a transaction compensation is *static*: the execution cannot change or modify it. In [62] the expressive power of $\texttt{web}\pi$ is shown by an encoding of the `scope` construct of `BPEL` [1]. Timing issues could be also considered in roll-π, where if a transaction runs out of time it will be automatically rolled-back. Moreover, time could also elapse while doing backward computation.

Expressiveness of long running transactions and of the way of composing compensations is thoroughly investigated in [60]. In this framework a

[5] Where ";" is the sequencing operator.

transaction has the following form:

$$t[P, Q]$$

where (as usual) t is the transaction name, P is the body of the transaction and Q is its compensation. As in $\texttt{web}\pi$ a transaction fails upon receiving the signal $\bar{t}$. This message can be generated internally, by the transaction itself, or by the enclosing environment. The authors distinguish three ways of constructing compensations: (*i*) *static* recovery, (*ii*) *parallel* recovery and (*iii*) *dynamic* recovery. In (*i*) compensations are static, that is the evolution of a transaction body cannot modify its compensation, meanwhile in (*ii*) and (*iii*) the body of a transaction can modify its compensation by means of a primitive that mimics a lambda abstraction to which the compensation is passed. Let us see an example of how such primitive works:

$$t[inst\lfloor \lambda X.R \rfloor.P, Q] \rightarrow t[P, R\{^{Q}/_{X}\}]$$

depending on the form of R both parallel and dynamic recovery can be expressed. For example, if $R = X \mid Q_1$ then the resulting transaction would be $Q \mid Q_1$ (representing a parallel compensation); if $R = a.(X \mid Q_1)$ then the resulting compensation would be $a.(Q \mid Q_1)$. In order to respond to a fail with the most recent (and suited) compensation, priority is given to the installation primitive. Hence, if a transaction receives a fail this fail has effect only if the transaction body does not contain updates. Naturally when a transaction fails, it disappears and its compensation is released. It is shown that there exists an encoding of parallel recoveries into static ones, but there does not exist an encoding of dynamic updates into static ones. Said otherwise, dynamic recovery is strictly more expressive.

Various models of composing compensations in different calculi (in terms of expressiveness) are studied in [20]. In this framework actions are *atomic*: either they commit or they abort. There is no way to observe an intermediate state of an action. Each action is bound with its compensation, and this binding is expressed in the following way:

$$A \div B$$

Actions are grouped into *sagas* of the form $\{P\}$ where P is a composition of actions. The simplest calculus expressed by this framework is the *sequential saga*, where actions can be just composed into sequences. As a saga evolves by executing its inner actions, its compensation is composed. For example:

$$\{A_1 \div B_1; A_2 \div B_2; A_3 \div B_3\ ,\ 0\}$$

after executing actions A_1, A_2 (they successfully committed) we obtain the following saga:

$$\{A_3 \div B_3\ ,\ B_2; B_1\}$$

where the compensation of the entire saga in case of A_3 fails is: the compensation of A_2 followed by the compensation of A_1. Let us note that if A_3 fails its compensation B_3 is not executed since B_3 serves to compensate A_3 only if it successfully commits. This is why actions are considered atomic, and then it is no possible to observe intermediate states of them. Compensations are executed in the reverse order in which actions have been executed. So, a saga involving $A_1, \ldots A_n$ activities is guaranteed to execute the entire sequence $A_1; \ldots; A_n$ (where to the action A_i corresponds the compensation B_i) or the compensated sequence $A_1; \ldots A_j; B_j; \ldots; B_1$ for some $j < n$. If a compensation fails, then the entire saga aborts, leaving the entire system in an undefined state. This implies that compensations cannot be sagas themselves, or that a compensation cannot have its own compensation. Different and more complex models of composing activities (e.g. parallel execution, nested sagas, programmable compensations) are explored, and each time the way compensations are composed and executed is formalized. Naturally, if we are able to write compensations that are the exact complement of the actions, that is $A_i \div A_i^{-1}$,then it is possible to write sagas whose compensations will imply the perfect rollback of each executed action, as in roll-π. Indeed, such kind of sagas can be easily encoded in roll-π: it is just sufficient that each compensation is a rollback operation aiming at reversing the corresponding action (communication). Moreover if all the actions are correlated, that is all the actions depend on the first one, then all the compensations will result in a rollback aiming at reversing the action A_1.

The need to distinguish generic faults from compensating requests in a distributed system led to the development of an extension of JOLIE [49,59,74]: a programming language for service oriented applications with primitives for dealing with fault management. A transactional scope has the form:

$$\mathtt{scope}_q(P, \mathcal{H})$$

where q is the name of the scope, P is the body and $\mathcal{H}$ is a mapping from scopes and faults to error handlers. When a scope successfully terminates, its handlers are automatically installed to the enclosing scope. In this way an action (scope) can always be compensated. There is a distinction between faults (generated by the primitive $\texttt{throw}(f)$) and a request of compensating a scope (generated by the primitive $\texttt{comp}(q)$). A fault f is propagated till there is a scope able to handle it (in a mechanism similar to Java when exceptions are raised), that is the scope defines $\mathcal{H}(f)$. During its propagation the fault kills all the scopes unable to handle it. In this way it is given the possibility to deal with several kind of faults. A request of compensation never kills a scope. A primitive for installing handlers is given (allowing so dynamic compensations) and it has the priority on fault handling. This mechanism (along with handlers installation) is similar to the one used in [60]. It would be nice to extend roll-π with mechanisms of dynamic compensations, such as the one proposed in [60] and JOLIE. In this way it would be possible, while undoing the computational history, to compensate irreversible actions or it would be possible to write compensations such as $\mathsf{roll}\ k; P$ whose semantics is: get the system back to the event k and then proceed with the execution of P.

A different formal approach to compensation composition is given in [34, 35] where the TransCCS calculus is introduced. TransCCS is an extension of CCS[6] with support for communicating transactions. Communicating transactions is a transactional model in which the *isolation* requirement is dropped. Such kind of transactions can be used to model automatic error recovery in distributed systems. In TransCCS a transaction takes the form:

$$[\![P \rhd_k Q]\!]$$

where P is the body, Q is the compensation and k is the name of a transaction. Aborting and committing are expressed by the following rules:

$$[\![co\ k \mid P \rhd_k Q]\!] \to P \qquad [\![P \rhd_k Q]\!] \to Q$$

where we can see that commit is explicitly expressed by the process $co\ k$, meanwhile a transaction can abort at any time. If a process gets into a transaction, then it is put in parallel with the transaction body and a copy of it is stored in the compensation part, according to the following rule:

$$[\![P \rhd_k Q]\!] \mid R \to [\![P \mid R \rhd_k Q \mid R]\!]$$

in this way if the transition k fails, all the processes that communicated with it are restored along with the original compensation. The way in which

[6] CCS with recursion.

processes get into a transaction and how something is stored in parallel with the transaction compensation, is similar to the enlarging scope mechanism of *cJoin* (as discussed above). The only restriction on this rule is that an absorbed process cannot contain the transaction name, that is a transaction cannot be directly committed by an external process. Naturally this rule should be used just to absorb processes able to communicate with the transaction body. The way of keeping a copy of the processes before a communication resembles the memory trick used in our $\rho\pi$ (see Chapter 3), and indeed it assures *perfect* rollback if correctly programmed. But unlike $\rho\pi$ where the memory mechanism permits to store processes related to a single communication, allowing so to roll-back just all the processes causally dependent by a particular communication, in TransCCS this is hard to program. An elegant way to encode perfect rollback in TransCCS is to exploit the recursion operator as in the following example:

$$\mu X.[\![P \triangleright_k X]\!]$$

in this way if after several interactions the transaction k fails, then its compensation will be composed by the transaction *itself* in parallel with all the processes that communicated with it, said otherwise on failure the system will roll-back exactly to a previous state. This model, even if simple and endowed with a good behavioural theory, has a few drawbacks: a transaction may *capture* processes that do not communicate with it, and then the only way to release them for the transaction is to (spontaneously) fail. Moreover failure is spontaneous, and it adds non determinism to the calculus.

2.2.3 Checkpointing and Rollback

The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. By exploiting these checkpoints, our rollback primitive is able to roll-back the entire system to an exact state indicated by the programmer. This is somehow akin to a checkpointing and roll-back schema. In distributed systems, checkpointing and roll-back (also known as checkpoint-recovery) is a technique of *backward* recovery (see [7]) for creating fault tolerant systems. The key concepts of this technique are: (i) periodic saves of system global state; (ii) in event of a fault, the state is restored via a rollback. This particular technique gives to a system (or to an application) the ability to save its state and tolerate faults by simply restoring an earlier state. In fact, when a checkpoint is executed, a *snapshot* of the entire system is taken and normally it is saved into some non-volatile medium. If a fault is detected, the recovery mechanism restores the system to the last checkpointed state.

There is an abundant literature (see [7, 37] for a quick review) on protocols and techniques on how to build a global (distributed) checkpoint, on what kind of information should represent the global state, and so on. We will address this topic by a programming language point of view, and then not considering libraries, middleware and operating system services.

Transient faults are unusual conditions that can be remedied by just re-executing the code which raised it. These faults are usually generated by temporally unavailable resources. For example, if a server is rebooting due to an internal error, then all the client requests issued during the rebooting time should be re-executed. In [101] a concurrent ML language, called *stabilizers*, for transient concurrent fault recovery in concurrent program is presented. The language introduces three new primitives: `stable`, `stabilize` and `cut`, able to deal with program global checkpoints. Primitive `stable` allows a thread to create a new stable section, that is a new global checkpoint. Primitive `stabilize`, issued by a single thread, allows the entire program to *roll-back* to the previous global checkpoint and finally primitive `cut` discards the current global checkpoint. This ensures that subsequent calls to `stabilize` will never cause the program to get back to a state that existed logically before the *cut*. Even if the semantics is proved to be *safe*, that is stabilization actions can never manufacture new states, the semantics cannot avoid the *domino effect*, that is a `stabilize` operation may unduly revert the program to a state beyond the target checkpoint. Thanks to the fine-grained causality tracking implied by our reversible substrate, our roll-π calculus does not suffer from uncontrolled cascading rollbacks (domino effect) which may arise with [101]. Nonetheless it is hard to make a comparison between our roll-π and the stabilizer programming language. Indeed, it would be necessary at least to introduce functions in our calculus, or to introduce our checkpointing mechanism in a functional concurrent language.

Bringing checkpoint-recovery technique in the actor model has been tackled by *Transactor* [41]: a fault tolerant programming model for composing loosely-coupled distributed components. It extends the actor model with constructs which distributed processes can use to to build globally consistent checkpoints. Basically a transactor can decide to commit its current state to a stable one. When a transactor becomes stable, further communications cannot change its state. It can be seen as a promise, to all the other transactors who communicated with it, that its state will not change. When an unstable transactor decides to roll-back, or it is asked to do so, it will cause the rollback of all the transactor whose state depended on the state of the unstable transactor. Interestingly the semantics also consider message loss. The language is proved to be sound, that is a trace containing node failures is equivalent to a normal one not containing failures, but possibly message losses. Moreover, checkpointing is possible just under certain conditions, and not in general cases. Indeed, not all the transactor programs can reach global checkpoints. A trivial program with a transactor that sends messages introducing dependencies, but never stabilizes or tries to checkpoint, will eliminate the ability of its dependent parties to reach checkpoints. Hence the authors introduces the *Universal Checkpoint Protocol* (UPC) that assumes a set of preconditions that will entail global checkpointing for a set T of transactors. Preconditions require that: (i) transactors in set T know their acquaintances indicated by the set ACQ and that this set is saved in their states; (ii) transactors in T eventually stabilize and start the UCP protocol, and moreover all the transactors need to be able to answer to `ping` messages; (iii) once one transactor in T stabilizes the others cannot rollback; (iv) during the UCP there cannot be failures. Finally, when a transactor in T stabilizes it:

1. Pings every transactor in ACQ.
2. Checks if it is dependent:
   - if not it pings every transactor in ACQ, checkpoints and ends the protocol;
   - if so, it pings every transactor in ACQ, checkpoints and waits for the incoming pings.
3. On reception of a ping message, goes back to 2.

Thanks to the fine-grained causality tracking implied by our reversible substrate, roll-π in contrast to transactor, provides a built-in guarantee that, in failure-free computations, rollback is always possible and reaches a consistent state (soundness of backward reduction). Hence we do not need to resort to preconditions or complex algorithms such as UPC to reach a consistent states.

From a formal point of view, various abstractions for fault-tolerant systems have been studied in [26]. In this work, a variant of π calculus is given, where processes can be aggregated in *conclaves* (groups). A conclave have the form:

$$c\{P_1 \mid \ldots \mid P_n\}$$

and each conclave is endowed with a log of the form

$$c\{\{L\}\}$$

where L is a collection of logical propositions. Conclaves have flat structures, that is there is no support for nesting, hence this model fails in modelling nested transactions. Processes can query logs (also those belonging to other conclaves) and can await the verification of certain conditions. But a process can just modify the log of its own conclave. Conclaves and logs are given as building blocks for fault tolerant distributed systems. If a process in a conclave fails, then the entire conclave fails and failure is propagated through causally dependent conclaves. Causal dependencies among conclaves are stored into logs. By varying log entries and rules for adding information to logs different transactional models can be easily encoded. For example a *compensation* for an aborting conclave can be easily written as a process that awaits for the *abort* state of a particular log, and then continues with the process in charge compensating the conclave. Nonetheless, this framework does not provides automatic supports for undoing the effects of conclaves that abort, while our roll-π directly provides a primitive to undo all the effects of a communication.

Chapter 3

The $\rho\pi$ calculus

In this chapter, we continue the study undertaken by Danos and Krivine, on RCCS, by tackling the following question: how can we introduce reversible actions in a higher-order concurrent calculus, specifically, the asynchronous Higher-Order π-calculus (HOπ)? This question finds its motivation in the pursuit of a suitable programming model for the construction of reliable and adaptive systems, and hence in the need to study the combination of reliable programming abstractions with modular dynamicity constructs enabling dynamic software update and on-line reconfiguration. As answer to this question, we define a reversible variant of the Higher-Order π-calculus (HOπ) [87]. A general method for reversing process calculi has been proposed by Phillips and Ulidowski in [82]. Unfortunately, it is only given for calculi whose operational semantics can be defined using SOS rules conforming to the *path* format, which is not the case for HOπ [77]. We therefore adopt an approach inspired by that of Danos and Krivine, but with significant differences. In particular, in their RCCS approach, the usual congruence laws associated with the parallel operator do not hold. Our first contribution is thus a simple syntax and reduction semantics for a reversible HOπ calculus called $\rho\pi$, with a novel way to define reversible actions while preserving the usual structural congruence laws of HOπ, notably the associativity and commutativity of the parallel operator.

The rest of the chapter is organized as follows: we first start with an informal discussion presenting the issues about reversing a calculus and making a comparison with the reversible CSS (RCCS); then we present the syntax and semantics of $\rho\pi$ (for reversible Higher-Order π) and its properties.

3.1 Informal Presentation

Building a reversible variant of a process calculus involves devising appropriate syntactic representations for computation histories. In general, since process calculi are not confluent and processes are non-deterministic, reversing a (forward) computation history means undoing the history not in a deterministic way, but in a *causally consistent* fashion, where states that are reached during a backward computation are states that could have been reached during the computation history by just performing independent actions in a different order. In RCCS, Danos and Krivine achieve this with CCS without recursion by attaching a memory m to each process P, in the monitored process construct $m : P$. A memory in RCCS is a stack of information needed for processes to backtrack. Thus, if two processes P_1 and P_2 can synchronize on a channel a to evolve into P_1' and P_2', respectively, then the parallel composition of monitored processes $m_1 : (P_1 + Q_1)$ and $m_2 : (P_2 + Q_2)$ can evolve, according to RCCS semantics, as follows:

$$m_1 : (P_1 + Q_1) \mid m_2 : (P_2 + Q_2) \to \langle m_2, a, Q_1 \rangle \cdot m_1 : P_1' \mid \langle m_1, a, Q_2 \rangle \cdot m_2 : P_2'$$

The memory $\langle m_2, a, Q_1 \rangle \cdot m_1$ represents the fact that the process tagged by this memory has performed a synchronization on channel a, interacting with the process with tag m_2, and discarding the alternative process Q_1.

Additionally, Danos and Krivine rely on the following structural congruence rule:

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

This rule ensures that each primitive thread, i.e. some process of the form $R_1 + R_2$, gets its own unique identity. Since this rule *stores* the exact position of a process in a parallel composition, it is not compatible with the usual structural congruence rules for the parallel operator, namely associativity, commutativity, and $\mathbf{0}$ as neutral element. Indeed RCCS is not equipped with the standard equivalence rules for the parallel operator, otherwise one could have that structurally equivalent processes may reduce to not structurally equivalent ones. For example, if we take the RCCS process $R = m : (a.P \mid \overline{a}.Q)$, by using standard equivalence for the parallel, one may write $S = m : (a.P \mid \overline{a}.Q \mid 0)$ with $R \equiv S$. Now, if we consider the execution of R and S we get:

$$\begin{aligned}
R \equiv & \langle 1 \rangle \cdot m : a.P \mid \langle 2 \rangle \cdot m : \overline{a}.Q \to \langle m_2, a, 0 \rangle \cdot m_1 : P \mid \langle m_1, \overline{a}, 0 \rangle \cdot m_2 : Q = R_1 \\
S \equiv & \langle 1 \rangle \cdot m : a.P \mid \langle 2 \rangle \cdot \langle 1 \rangle \cdot m : \overline{a}.Q \mid \langle 2 \rangle \cdot \langle 2 \rangle \cdot m : 0 \to \\
& \quad \langle m_{2,1}, a, 0 \rangle \cdot m_1 : P \mid \langle m_1, \overline{a}, 0 \rangle \cdot \langle 2 \rangle \cdot \langle 1 \rangle \cdot m : Q \mid \langle 2 \rangle \cdot \langle 2 \rangle \cdot m : 0 = S_1
\end{aligned}$$

with $m_i = \langle i \rangle \cdot m$, $m_{i,j} = \langle i \rangle \cdot \langle j \rangle \cdot m$ and $i, j \in \{1, 2\}$. But, as one can see, we have that $R \equiv S$, $R \to R_1$, $S \to S_1$ but $R_1 \not\equiv S_1$.

Danos and Krivine suggest that it could be possible to work up to tree isomorphisms on memories, but this would indeed lead to a more complex syntax, as well as additional difficulties (see Remark 3.4 below).

We adopt for $\rho\pi$ a different approach: instead of associating each thread with a stack that records, essentially, past actions and positions in parallel branches, we rely on simple thread tags, which act as unique identifiers but have little structure, and on new process terms, which we call *memories*, which are dedicated to undoing a single (forward) computation step. Note that in RCCS memories of monitored processes are used also as identifiers since they are unique among them.

To ease the understanding of our approach we will first (informally) introduce HOπ and then we will show how to hone the HOπ reduction rule in order to reverse it. The Higher-Order π-calculus is a variant of the π-calculus where the data exchanged by processes are processes themselves. The syntax of HOπ is depicted in Figure 3.1 (page 44) by just considering the productions of processes P. A communication in HOπ is given by the following rule:

$$a\langle P \rangle \mid a(X) \triangleright Q \to Q\{^P/_X\}$$

where a message $a\langle P \rangle$ on channel a is read by a receiver process (or *trigger*) $a(X) \triangleright Q$. The result of the message receipt consists in the launch of an instance $Q\{^P/_X\}$ of the body of the trigger Q, with the formal parameter X instantiated by the received value, i.e. process P. After the communication has been performed, from process $Q\{^P/_X\}$ there is no information about the pair of processes (input, output) that generated it and hence it is impossible to reverse this communication. In order to reverse a communication, we need to add information about the previous configuration. We then modify the above rule, by labelling each process with an identifier (*tag*), and by storing the pair of processes that give rise to a communication into a *memory process*. Hence, a *forward* computation step in $\rho\pi$ (noted with arrow $\twoheadrightarrow$) takes the following form:

$$(\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k.\, k : Q\{^P/_X\} \mid [M; k]$$

with $M = (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q)$. Each thread (message and trigger) participating in the above computation step is uniquely identified by a tag: κ_1 identifies the message $a\langle P \rangle$, and κ_2 identifies the trigger $a(X) \triangleright Q$. The result of the message receipt consists in a classical part and two side effects.

The classical part is similar to the HOπ, where the continuation of the trigger is instantiated with the received parameter. The two side effects are: (i) the tagging of the newly created process $Q\{^P/_X\}$ by a fresh name k (operator ν is the standard restriction operator of the π-calculus), and (ii) the creation of a memory process $[M; k]$. M is simply the configuration on the left hand side of the reduction. Note that by using the restriction operator, we enforce the property that processes are uniquely identified.

In this setting, a *backward* computation step takes the form of an interaction between a memory and a process tagged with the appropriate name: when a memory $[M; k]$ is put in presence of a process tagged with k, a *backward* reduction (noted with the arrow $\rightsquigarrow$) can take place. It kills the process tagged with k and reinstates the configuration M:

$$(k : P) \ \mid \ [M; k] \ \rightsquigarrow \ M$$

We thus have:

$$M \ \twoheadrightarrow \ \nu k.\, k : Q\{^P/_X\} \mid [M; k] \ \rightsquigarrow \ \nu k.\, M$$

Since k is fresh, $\nu k.\, M$ is actually (structurally) equivalent to M. We thus have a perfect reversal of a forward computation: $M \twoheadrightarrow\rightsquigarrow M$.

To ensure that unique identifiers are generated (and preserved) through a parallel composition we adopt a different technique from the one of [29]: we *flatten* the structure of a parallel composition into a composition of *primitive* threads, that is a composition of messages and triggers, and we create a sequence of new names. Then to each primitive thread is given a new *structured* identifier containing information about the generated sequence and the initial identifier of the parallel composition. For example:

$$\begin{aligned}&k_1 : (a\langle 0\rangle \mid b\langle 0\rangle \mid a(X) \triangleright Q) \equiv \\ &\nu\tilde{h}.\, (\langle h_1, \tilde{h}\rangle \cdot k : a\langle 0\rangle) \mid (\langle h_2, \tilde{h}\rangle \cdot k : b\langle 0\rangle) \mid (\langle h_3, \tilde{h}\rangle \cdot k : a(X) \triangleright Q)\end{aligned}$$

with $\tilde{h} = \{h_1, h_2, h_3\}$. An identifier of the form $\langle h_i, \tilde{h}\rangle \cdot k$ essentially tells us that the sequence $\tilde{h}$ has been generated by a parallel process identified by k, containing $|\tilde{h}|$ primitive threads. With this simple trick associativity and transitivity of the parallel operator are preserved.

Remark 3.1 *Following Danos and Krivine [30], one could consider also taking into account* irreversible *actions. We do not do so in this document, though, because we focus on reversibility, and because adding irreversible actions to $\rho\pi$ would be conceptually straightforward.*

3.2 Syntax and Semantics

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, and the set $\mathcal{V}$ of *process variables*. The set $\mathcal{I} = \mathcal{N} \cup \mathcal{K}$ is called the set of *identifiers*. We denote by $\mathbb{N}$ the set of natural integers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{I}$; X, Y, Z range over $\mathcal{V}$. We denote by $\tilde{u}$ a finite set of identifiers $\{u_1, \ldots, u_n\}$.

Syntax. The syntax of the $\rho\pi$ calculus is given in Figure 3.1 (in writing $\rho\pi$ terms, we freely add balanced parenthesis around terms to disambiguate them). *Processes* of the $\rho\pi$ calculus, given by the P, Q productions in Figure 3.1, are the standard processes of the asynchronous Higher-Order π-calculus. A receiver process (or *trigger*) in $\rho\pi$ takes the form $a(X) \triangleright P$. Messages have no continuation, since $\rho\pi$ is an asynchronous calculus. $P \mid Q$ is the parallel composition of P and Q, while $\nu a.\, P$ binds name a inside process P.

Processes in $\rho\pi$ cannot directly execute, only *configurations* can. *Configurations* in $\rho\pi$ are given by the M, N productions in Figure 3.1. A configuration is built up from *threads* and *memories*.

A *thread* $\kappa : P$ is just a tagged process P, where the tag κ is either a single key k or a pair of the form $\langle h, \tilde{h} \rangle \cdot k$, where $\tilde{h}$ is a set of keys and $h \in \tilde{h}$. A tag serves as an identifier for a process. As we will see below, together with memories tags help to capture the flow of causality in a computation.

A *memory* is a process of the form $[\mu; k]$, which keeps track of the fact that a configuration μ was reached during execution, that triggered the launch of a thread tagged with the fresh tag k. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory, and k the *tag* of the memory. Memories are generated by computation steps and are used to reverse them. The configuration part $\mu = (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q)$ of the memory records the message $a\langle P \rangle$ and the trigger involved in the message receipt $a(X) \triangleright Q$, together with their respective thread tags κ_1, κ_2.

We note $\mathcal{P}$ the set of $\rho\pi$ processes, and $\mathcal{C}$ the set of $\rho\pi$ configurations. We call *agent* an element of the set $\mathcal{A} = \mathcal{P} \cup \mathcal{C}$. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$; L, M, N range over $\mathcal{C}$; and A, B, C range over $\mathcal{A}$. We call *primitive thread process* a process that is either a message $a\langle P \rangle$ or a trigger $a(X) \triangleright P$. We let τ and its decorated variants range over primitive thread processes.

Remark 3.2 *There is no construct for recursive definitions or replicated*

$$
\begin{array}{rlr}
P, Q ::= & & \textit{processes} \\
& \mathbf{0} & \textit{null process} \\
& \mid\ X & \textit{process variable} \\
& \mid\ \nu a.\, P & \textit{restriction} \\
& \mid\ (P \mid Q) & \textit{parallel} \\
& \mid\ a\langle P \rangle & \textit{message} \\
& \mid\ a(X) \triangleright P & \textit{trigger} \\
\\
M, N ::= & & \textit{configurations} \\
& \mathbf{0} & \textit{null configuration} \\
& \mid\ \nu u.\, M & \textit{restriction} \\
& \mid\ (M \mid N) & \textit{parallel} \\
& \mid\ \kappa : P & \textit{thread} \\
& \mid\ [\mu; k] & \textit{memory} \\
\\
\kappa ::= & k \ \mid\ \langle h, \tilde{h} \rangle \cdot k & \textit{tags} \\
\mu ::= & ((\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q)) & \textit{configuration part} \\
\\
u \in & \mathcal{I} & \\
a \in & \mathcal{N} & \\
X \in & \mathcal{V} & \\
h, k \in & \mathcal{K} & \\
\kappa \in & \mathcal{T} &
\end{array}
$$

Figure 3.1: Syntax of $\rho\pi$.

processes in $\rho\pi$. This is because a replicated process can be easily defined[1] *in $\rho\pi$: let $\mathtt{B} = (a(X) \triangleright a\langle X\rangle \mid X)$, then $\mathtt{Bang}_P = \nu a.\, a\langle P \mid \mathtt{B}\rangle \mid \mathtt{B}$ provides any number of copies of P.*

Free names and free variables. Notions of free identifiers and free (process) variables in $\rho\pi$ are classical. It suffices to note that constructs with binders are of the forms: $\nu a.\, P$ which binds the name a with scope P; $\nu u.\, M$, which binds the identifier u with scope M; and $a(X) \triangleright P$, which binds the variable X with scope P. We denote by $\mathtt{fn}(P)$, $\mathtt{fn}(M)$ and $\mathtt{fn}(\kappa)$ the set of free names, free identifiers, and free keys, respectively, of process P, of configuration M, and of tag κ. Note in particular that $\mathtt{fn}(\kappa : P) = \mathtt{fn}(\kappa) \cup \mathtt{fn}(P)$, $\mathtt{fn}(k) = \{k\}$ and $\mathtt{fn}(\langle h, \tilde{h}\rangle \cdot k = \{h\} \cup \tilde{h} \cup \{k\}$. We say that a process P or a configuration M is closed if it has no free (process) variable. We denote by $\mathcal{P}^{cl}$ the set of closed processes, $\mathcal{C}^{cl}$ the set of closed configurations, and $\mathcal{A}^{cl}$ the set of closed agents.

Remark 3.3 *In the remainder of this chapter, we adopt* Barendregt's Variable Convention*: If terms $t_1, \ldots, t_n$ occur in a certain context (e.g. definition, proof), then in these terms all bound identifiers and variables are chosen to be different from the free ones.*

Consistent configurations. Not all configurations allowed by the syntax in Figure 3.1 are meaningful. In a memory $[\mu; k]$, tags occurring in the configuration part μ must be different from the tag k. This is because the key k is freshly generated when a computation step (a message receipt) takes place, and it is used to identify the newly created thread. Tags appearing in the configuration part identify threads (message and trigger) which have participated in the computation step. In a configuration M, we require all the threads to be uniquely identified by their tag, and we require consistency between threads and memories: if M contains a memory $[N; k]$ (i.e. $[N; k]$ occurs as a sub-term of M), we require M to also contain a thread tagged with k: components of this thread, i.e. threads whose tags have k as a suffix, can occur either directly in parallel with $[N; k]$ or in the configuration part of another memory contained in M (because they may have interacted with other threads). We call *consistent* a configuration that obeys these syntactic constraints. We defer the formal definition of consistent configurations to Section 3.2.1.

[1] This is a basic property of all higher-order calculi with name creation.

$$(\text{E.}\textsc{ParC})\ A \mid B \equiv B \mid A \qquad (\text{E.}\textsc{ParA})\ A \mid (B \mid C) \equiv (A \mid B) \mid C$$

$$(\text{E.}\textsc{NilM})\ A \mid \mathbf{0} \equiv A \qquad (\text{E.}\textsc{NewN})\ \nu u.\, \mathbf{0} \equiv \mathbf{0}$$

$$(\text{E.}\textsc{NewC})\ \nu u.\, \nu v.\, A \equiv \nu v.\, \nu u.\, A \qquad (\text{E.}\textsc{NewP})\ (\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$$

$$(\text{E.}\alpha)\ A =_\alpha B \implies A \equiv B \qquad (\text{E.}\textsc{TagN})\ \kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$$

$$(\text{E.}\textsc{TagP})\ k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\, \prod_{i=1}^{n} (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Figure 3.2: Structural congruence for $\rho\pi$.

3.2.1 Operational semantics

The operational semantics of the $\rho\pi$ calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over closed configurations $\rightarrow \subset \mathcal{C}^{cl} \times \mathcal{C}^{cl}$, and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations $\equiv \subset \mathcal{P}^2 \cup \mathcal{C}^2$. We define evaluation contexts as "configurations with a hole $\cdot$" given by the following grammar:

$$\mathbb{E} ::= \cdot \ \mid \ (M \mid \mathbb{E}) \ \mid \ \nu u.\, \mathbb{E}$$

General contexts $\mathbb{C}$ are just processes or configurations with a hole. A congruence on processes and configurations is an equivalence relation $\mathcal{R}$ that is closed for general contexts: $P \,\mathcal{R}\, Q \implies \mathbb{C}[P] \,\mathcal{R}\, \mathbb{C}[Q]$ and $M \,\mathcal{R}\, N \implies \mathbb{C}[M] \,\mathcal{R}\, \mathbb{C}[N]$.

The relation $\equiv$ is defined as the smallest congruence on processes and configurations that satisfies the rules in Figure 3.2. We note $t =_\alpha t'$ when terms t, t' are equal modulo α-conversion. If $\tilde{u} = \{u_1, \ldots, u_n\}$, then $\nu \tilde{u}.\, A$ stands for $\nu u_1.\, \ldots \nu u_n.\, A$. We note $\prod_{i=1}^{n} A_i$ for $A_1 \mid \ldots \mid A_n$ (there is no need to indicate how the latter expression is parenthesized because the parallel operator is associative by rule E.ParA). In rule E.TagP, processes τ_i are primitive thread processes. Recall the use of the variable convention in these rules: for instance, in the rule $(\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$ the variable convention makes implicit the condition $u \notin \texttt{fn}(B)$. The structural congruence rules are the usual rules for the π-calculus (E.ParC to E.α) without the rule dealing with replication, and with the addition of two new rules dealing with tags: E.TagN and E.TagP. Rule E.TagN is a scope extrusion rule to push restrictions to the top level. Rule E.TagP allows to generate unique tags for each primitive thread process in a configuration, by spreading the information of a tag among all the primitive processes in parallel. For example, the process $k : (a\langle P\rangle \mid b\langle Q\rangle \mid (a(X) \triangleright R))$ is structurally equivalent to:

$$\nu \tilde{h}. \,(\langle h_1, \tilde{h}\rangle \cdot k : a\langle P\rangle) \mid (\langle h_2, \tilde{h}\rangle \cdot k : b\langle Q\rangle) \mid (\langle h_3, \tilde{h}\rangle \cdot k : a(X) \triangleright R)$$

with $\tilde{h} = \{h_1, h_2, h_3\}$.

An easy induction on the structure of terms provides us with a kind of normal form for configurations (by convention $\prod_{i\in I} A_i = \mathbf{0}$ if $I = \emptyset$):

Lemma 3.1 (Thread normal form) *For any configuration M, we have*

$$M \equiv \nu \tilde{u}. \prod_{i\in I} (\kappa_i : \rho_i) \mid \prod_{j\in J} [M_j : k_j]$$

with $\rho_i = \mathbf{0}$, $\rho_i = a_i\langle P_i\rangle$, or $\rho_i = a_i(X_i) \triangleright P_i$.

We say that a binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$(\text{R.Ctx}) \ \frac{M \mathrel{\mathcal{R}} N}{\mathbb{E}[M] \mathrel{\mathcal{R}} \mathbb{E}[N]}$$

$$(\text{R.Eqv}) \ \frac{M \equiv M' \qquad M' \mathrel{\mathcal{R}} N' \qquad N' \equiv N}{M \mathrel{\mathcal{R}} N}$$

The reduction relation $\to$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the *backward* reduction relation $\rightsquigarrow$: $\to = \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined to be the smallest evaluation-closed binary relations on closed configurations satisfying the rules in Figure 3.3 (note again the use of the variable convention: in rule R.Fw the key k is fresh). We note $\Rightarrow$ the reflexive and transitive closure of $\to$.

$$(\text{R.Fw}) \ \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \twoheadrightarrow \nu k. \,(k : Q\{^{P}/_{X}\}) \mid [\mu; k]}$$

$$(\text{R.Bw}) \quad (k : P) \mid [M; k] \rightsquigarrow M$$

Figure 3.3: Reduction rules for $\rho\pi$.

The rule for forward reduction (R.Fw) is the standard communication rule of the Higher-Order π-calculus with two side effects: (i) the creation of a new memory to record the configuration that gave rise to it, namely the parallel composition of a message and a trigger, properly tagged (tags κ_1 and κ_2 in the rule); (ii) the tagging of the continuation of the message receipt (with the fresh key k, enforcing so the fact that tags are unique). The rule for backward reduction (R.Bw) is straightforward: in presence of the thread tagged with key k, memory $[M; k]$ reinstates the configuration M that gave rise to the tagged thread. With the reduction rules and the structural laws in place, we can see how the structural rule E.TagP is used by the reduction. In particular the rule, if it is used from left to right after a forward step, lets the continuation of a trigger (if it is a parallel composition) continue executing in the forward direction. On the other side, when used from right to left, E.TagP gathers back all the primitive processes belonging to the same parallel composition identified by a particular tag. An example of execution will make it clear. Let $M = (k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright b\langle X\rangle \mid b(X) \triangleright 0)$, we have that:

$$\begin{aligned}
&M \twoheadrightarrow \nu k.\, k : (b\langle P\rangle \mid b(X) \triangleright 0) \mid [M; k] &(1)\\
&\equiv \nu k, h_1, h_2.\, (\langle h_1, \tilde{h}\rangle \cdot k : b\langle P\rangle)) \mid (\langle h_2, \tilde{h}\rangle \cdot k : b(X) \triangleright 0) \mid [M; k] &(2)\\
&\twoheadrightarrow \nu k, h_1, h_2, k_3.\, (k_3 : 0) \mid [M; k] \mid [M_1; k_3] &(3)\\
&\rightsquigarrow \nu k, h_1, h_2.\, (\langle h_1, \tilde{h}\rangle \cdot k : b\langle P\rangle)) \mid (\langle h_2, \tilde{h}\rangle \cdot k : b(X) \triangleright 0) \mid [M; k] &(4)\\
&\equiv \nu k.\, k : (b\langle P\rangle \mid b(X) \triangleright 0) \mid [M; k] &(5)
\end{aligned}$$

with $\tilde{h} = \{h_1, h_2\}$, $M_1 = (\langle h_1, \tilde{h}\rangle \cdot k : b\langle P\rangle)) \mid (\langle h_2, \tilde{h}\rangle \cdot k : b(X) \triangleright 0)$. We can note in (2) the use of the rule E.TagP from left to right, in order to allow the two primitive processes to execute (3). On the other side, we use the rule in the opposite way in (5) in order to build back the parallel composition that generated the two single primitive processes.

Remark 3.4 *One could have thought of mimicking the structural congruence rule dealing with parallel composition in [29] (RCCS), using a monoid structure for tags:*

$$(\text{E.TagP}^\bullet)\ \kappa : (P \mid Q) \equiv \nu h_1, h_2.\, (h_1 \cdot \kappa : P) \mid (h_2 \cdot \kappa : Q)$$

Unfortunately using E.TagP$^\bullet$ *instead of* E.tagP *would introduce some undesired non-determinism, which would later complicate our definitions (in relation to causality) and proofs. For instance, let* $M = k : a\langle Q\rangle \mid (h : a(X) \triangleright X)$. *We have:* $M \rightarrow M' = \nu l.\, (l : Q) \mid [M; l]$ *Now, assuming* E.tagP$^\bullet$, *we would have*

$$M \equiv (k : (a\langle Q\rangle \mid \mathbf{0})) \mid (h : a(X) \triangleright X) \equiv \nu h_1, h_2. ((h_1 \cdot k : a\langle Q\rangle) \mid (h_2 \cdot k : \mathbf{0})) \mid (h : a(X) \triangleright X)$$

Let $M_1 = (h_1 \cdot k : a\langle Q\rangle) \mid (h : a(X) \triangleright X)$*. We would then have:* $M \to M''$*, where* $M'' = \nu h_1, h_2, l. (l : Q) \mid [M_1; l] \mid (h_2 \cdot k : \mathbf{0})$*. Clearly* $M' \not\equiv M''$*, which means that a seemingly deterministic configuration,* M*, would have in fact two (actually, an infinity) of derivations towards non structurally equivalent configurations. By insisting on tagging only primitive thread processes,* E.TagP *avoids this unfortunate situation.*

We can characterize this by proving a kind of determinacy *lemma for* $\rho\pi$*, which fails if we replace rule* E.TagP *with rule* E.TagP$^{\bullet}$*. Extend the grammar of* $\rho\pi$ *with marked primitive thread processes of the form* $\tau^{\bullet}$*. This extended calculus has exactly the same structural congruence and reduction rules than* $\rho\pi$*, but with possibly marked primitive thread processes. Now call* primed *a closed configuration* M *with exactly two marked processes of the form* $a\langle P\rangle^{\bullet}$ *and* $(a(X) \triangleright Q)^{\bullet}$*. Anticipating the definition of the reduction relation* $\to$ *below, we have:*

Lemma 3.2 (Determinacy) *Let* M *be a primed configuration such that* $M \equiv M_1 = \mathbb{E}_1[\kappa_1 : a\langle P\rangle^{\bullet} \mid \kappa_2 : (a(X) \triangleright Q)^{\bullet}]$ *and* $M \equiv M_2 = \mathbb{E}_2[\kappa'_1 : a\langle P\rangle^{\bullet} \mid \kappa'_2 : (a(X) \triangleright Q)^{\bullet}]$*. Assume* $M_1 \to M'_1$ *and* $M_2 \to M'_2$ *are derived by applying* R.Fw *with configurations* $\kappa_1 : a\langle P\rangle^{\bullet} \mid \kappa_2 : (a(X) \triangleright Q)^{\bullet}$*, and* $\kappa'_1 : a\langle P\rangle^{\bullet} \mid \kappa'_2 : (a(X) \triangleright Q)^{\bullet}$ *respectively, followed by* R.Ctx*. Then* $M'_1 \equiv M'_2$*.*

Proof. *By induction on the form of* $\mathbb{E}_1$*, and case analysis on the form of* κ_1 *and* κ_2*.* □

We can now formally define the notion of *consistent configuration.*

Definition 3.1 (Consistent configuration)
A configuration $M \equiv \nu\tilde{u}. \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[M_j; k_j]$*, with* $\rho_i = \mathbf{0}$ *or* ρ_i *a primitive thread process,* $M_j = \delta_j : R_j \mid \gamma_j : T_j$*,* $R_j = a_j\langle P_j\rangle$*,* $T_j = a_j(X_j) \triangleright Q_j$*, is said to be* consistent *if the following conditions are met:*

1. *For all* $j \in J$*,* $k_j \neq \delta_j$*,* $k_j \neq \gamma_j$ *and* $\delta_j \neq \gamma_j$
2. *For all* $i_1, i_2 \in I$*,* $i_1 \neq i_2 \implies \kappa_{i_1} \neq \kappa_{i_2}$
3. *For all* $i \in I, j \in J$*,* $\kappa_i \neq \delta_j$ *and* $\kappa_i \neq \gamma_j$
4. *For all* $j_1, j_2 \in J$*,* $j_1 \neq j_2 \implies \{\delta_{j_1}, \gamma_{j_1}\} \cap \{\delta_{j_2}, \gamma_{j_2}\} = \emptyset$

5. *For all* $j_1, j_2 \in J$, $j_1 \neq j_2 \implies k_{j_1} \neq k_{j_2}$

6. *For all* $j \in J$, *there exist* $E \subseteq I$, $D \subseteq J \setminus \{j\}$, $G \subseteq J \setminus \{j\}$, *such that:*

$$\nu\tilde{u}.\, k_j : Q_j\{^{P_j}/_{X_j}\} \equiv \nu\tilde{u}.\, \prod_{e \in E} \kappa_e : \rho_e \mid \prod_{d \in D} \delta_d : R_d \mid \prod_{g \in G} \gamma_g : T_g$$

Roughly, consistent configurations enjoy two properties: (i) uniqueness of keys and (ii) that for each memory $[M; k]$ there are processes in the configuration corresponding to the continuation of M. In more detail, condition 1 ensures that the thread tag of a memory never occurs in its own configuration part. Condition 2 ensures that different threads are tagged by different keys. Conditions 3 and 4 are similar, but they concern threads inside memories too. Condition 5 states that all thread tags of memories are distinct. Condition 6 is the most tricky one. It requires that for each memory $[\kappa_{j_1} : a_j\langle P_j\rangle \mid \kappa_{j_2} : a_j(X_j) \triangleright Q_j; k_j]$ there are threads in the configuration whose composition gives the continuation $\nu\tilde{u}.\, k_j : Q_j\{^{P_j}/_{X_j}\}$. Note that because of the use of $\equiv$ there are only two possibilities: either there is a unique thread corresponding to the continuation tagged with k_j, or there are many of them, all tagged with complex keys having k_j as a suffix, generated by one application of rule E.TagP. These threads may be at top level or inside the configuration parts of other memories (as participants to other communications).

Consistency is a global property, so the composition of consistent configuration may lead to a non consistent configuration. To better understand consistency let us consider a few examples:

$$k_1 : a\langle 0\rangle \mid [(k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright P); k] \tag{1}$$
$$k : a\langle 0\rangle \mid [(k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright P); k] \tag{2}$$
$$k : a\langle 0\rangle \mid [(k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright a\langle X\rangle); k] \tag{3}$$
$$\nu\tilde{h}.\, (\langle h_1, \tilde{h}\rangle \cdot k : a\langle 0\rangle) \mid (k : b\langle 0\rangle) \tag{4}$$

configuration (1) is not consistent since it violates condition 3 on key k_1 and condition 6 on the memory. Configuration (2) is not consistent because it violates condition 6 (we assume $P \neq a\langle 0\rangle$). Configuration (3) is consistent since all the keys are unique and $k : a\langle X\rangle\{^0/_X\} \equiv k : a\langle 0\rangle$. Configuration (4) is also consistent, even if this may appear weird because of the key $\langle h_1, \tilde{h}\rangle \cdot k$. This is possible since we do not impose a well-formed condition on keys that are not generated by a memory (that is keys that are not memory thread tags). For this kind of keys we just require uniqueness, so we have that

$\langle h_1, \tilde{h}\rangle \cdot k \neq k$, and the configuration is consistent. To avoid this kind of badly formed keys, when needed, we will consider configurations generated by an initial configuration. We will postpone this discussion to Chapter 5.

Consistent configurations are preserved by reduction:

Lemma 3.3 *Let M be a consistent configuration. If $M \to N$ then N is a consistent configuration.*

We need a few auxiliary lemmas to prove this result. The lemma below gives a syntactic characterization of forward reductions.

Lemma 3.4 *Let M, N be configurations. Then $M \twoheadrightarrow N$ iff $M \equiv M'$ and $N' \equiv N$ with:*

$$M' = \nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j]$$

$$N' = \nu\tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j]$$

Proof: Let us start with the *if* direction. The proof is by induction on the derivation of the reduction $\twoheadrightarrow$. We have a case analysis on the last applied rule:

R.Fw: By hypothesis $M = \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q$ and $M \twoheadrightarrow \nu k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] = N$. The thesis follows immediatly by choosing $M' = M$ and $N' = N$.

R.Eqv: The thesis follows by transitivity of structural equivalence.

R.Ctx: The case of the empty context is banally verified by inductive hypothesis. If the context is a restriction then we have that $\nu u.\, M \twoheadrightarrow \nu u.\, N$ with $M \twoheadrightarrow N$ as hypothesis. By inductive hypothesis $M \equiv M'$ and $N' \equiv N$, with:

$$M' = \nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j]$$

$$N' = \nu\tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j]$$

Then

$$\begin{aligned}
&\nu u.\, M \equiv \nu u, \tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j] \\
&\nu u.\, N' = \nu u, \tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \\
&\qquad \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j]
\end{aligned}$$

with $\nu u.\, N' \equiv \nu u.\, N$, as desired.

For (left) parallel context we have that $M_1 \mid M \twoheadrightarrow M_1 \mid N$ with $M \twoheadrightarrow N$ as hypothesis. By inductive hypothesis $M \equiv M'$ with:

$$\begin{aligned}
&M' = \nu \tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j] \\
&N' = \nu \tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \\
&\qquad \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j]
\end{aligned}$$

with $N' \equiv N$. Also, from Lemma 3.1 $M_1 \equiv \nu \tilde{v}.\, \prod_{i\in I'} (\kappa_i : \rho_i) \mid \prod_{j\in J'} [M_j : k_j]$. Then

$$\begin{aligned}
&M_1 \mid M \equiv \nu \tilde{u}, \tilde{v}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I\cup I'} \kappa_i : \rho_i \mid \prod_{j\in J\cup J'} [\mu_j ; k_j] \\
&N'' = \nu \tilde{u}, \tilde{v}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \\
&\qquad \prod_{i\in I\cup I'} \kappa_i : \rho_i \mid \prod_{j\in J\cup J'} [\mu_j ; k_j]
\end{aligned}$$

with $N'' \equiv M_1 \mid N$ as desired. The case of right parallel context is similar.

For the other direction, note that the desired reduction can be derived by applying rule R.Fw followed by R.Ctx to derive

$$\begin{aligned}
&\nu \tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j] \twoheadrightarrow \\
&\nu \tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j ; k_j]
\end{aligned}$$

The thesis then follows by applying rule R.Eqv. □

The following lemma is similar to Lemma 3.4, but it considers backward reductions.

Lemma 3.5 *Let M, N be configurations. Then $M \rightsquigarrow N$ iff $M \equiv M'$ with $M' = \nu\tilde{u}, k.\, k : R \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} m_j$ and $\nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; k_j] = N'$ and $N \equiv N$.*

Proof: The proof is similar to the proof of Lemma 3.4 □

We can finally prove Lemma 3.3.

Proof of Lemma 3.3: By case analysis on the derivation of $M \rightarrow N$.

Let us consider the case $M \twoheadrightarrow N$, by Lemma 3.4 we have $M \equiv M'$ with $M' = \nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; k_j]$ and $\nu\tilde{u}, k.\, k : Q\{{}^{P}/_{X}\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; k_j] = N'$ and $N' \equiv N$. By hypothesis M is a consistent configuration so also M' is a consistent configuration (since the consistency is itself defined using $\equiv$ it is banally preserved by $\equiv$). Now we have to prove that N' is a valid configuration. The first five properties of validity check uniqueness of keys. The first property holds since, for memories already in M' it is part of the hypothesis, while for the new memory $[\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k]$ k is fresh and κ_1 and κ_2 are different by hypothesis (condition 2). The second condition is satisfied since the new tag k is fresh and applying axiom E.TagP also generates fresh distinct tags. Let us consider the third condition. For the new tag k it is verified since k is fresh and does not occur in $\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q$. For the new memory the condition is satisfied with respect to old thread processes thanks to the hypothesis (condition 2). The fourth condition is verified by hypothesis for threads inside old memories and thanks to second and third conditions for the new memory. The fifth condition is verified by hypothesis for the old keys, and since the key is freshly generated for the new one. The last condition holds for the new memory $[\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k]$, since we have that:

$$\nu\tilde{u}, k.\, k : Q\{{}^{P}/_{X}\} \equiv \nu\tilde{u}, k.\, k : Q\{{}^{P}/_{X}\}$$

and by hypothesis it holds for other memories. Let us note that the condition on memories that generated the two threads, identified by κ_1 and κ_2, participating to the last communication still holds, since the two threads are just moved from an active context to a memory.

The case $M \rightsquigarrow N$ is similar to the previous one, using Lemma 3.5 instead of Lemma 3.4. □

Remark 3.5 *The presented semantics and machinery for reversing HOπ can be easily adapted to define reversibility in first order π-calculus. In* *general, the combination of memories and identifiers should be enough to define reversibility in calculi with implicit substitutions. Indeed, the need for memories stems from the fact that substitution is not a bijective, hence irreversible, function: the only way to reverse a substitution is to record the exact form of the process before applying it.*

3.3 Basic properties of reduction in $\rho\pi$

In this section we show two main properties of $\rho\pi$: (i) that $\rho\pi$ is a conservative extension of HOπ and (ii) that each $\rho\pi$ reduction can be reversed.

We first recall HOπ syntax and semantics. The syntax of HOπ processes coincides with the syntax of $\rho\pi$ processes in Figure 3.1 (but HOπ has no concept of configuration). HOπ structural congruence, denoted $\equiv_\pi$, is the least congruence generated by rules in Figure 3.2 but E.TagN and E.TagP (restricted to processes). HOπ reduction relation $\rightarrow_\pi$ is the least evaluation-closed binary relation on closed processes defined by the rule:

$$(\text{HO.Red}) \quad a\langle P\rangle \mid a(X) \triangleright Q \rightarrow_\pi Q\{^P/_X\}$$

In order to show (i) we define the *erasing function* $\gamma : \mathcal{C} \rightarrow \mathcal{P}$, which maps a $\rho\pi$ configuration on its underlying HOπ process.

Definition 3.2 (Erasing function) *The* erasing function $\gamma : \mathcal{C} \rightarrow \mathcal{P}$ *is defined inductively by the following clauses:*

$$\begin{array}{lll} \gamma(\mathbf{0}) = \mathbf{0} & \gamma(\nu a.\, M) = \nu a.\, \gamma(M) & \gamma(\nu k.\, M) = \gamma(M) \\ \gamma(M \mid N) = \gamma(M) \mid \gamma(N) & \gamma(\kappa : P) = P & \gamma([M; k]) = \mathbf{0} \end{array}$$

Let us note that γ directly deletes the creation of new keys (νk) since this kind of names has no meaning into HOπ. Moreover it deletes all the extra machinery (tags and memories) used to reverse HOπ.

Lemma 3.7 below shows that $\rho\pi$ forward computations are indeed decorations on HOπ reductions. We first prove an auxiliary result relating $\rho\pi$ and HOπ structural congruences.

Lemma 3.6 *For all closed configuration M, N if $M \equiv N$ then $\gamma(M) \equiv_\pi \gamma(N)$.*

Proof: It is enough to prove that the thesis holds for each axiom (since γ is defined by structural induction). We have a case for each axiom. For the rules E.ParC, E.ParA, E.NilM, E.NewN, E.NewC, E.NewP and E.α

there is a corresponding rule in HOπ. Rules E.TagN and E.TagP instead reduce to the identity. □

Lemma 3.7 *For all configurations M, N, if $M \twoheadrightarrow N$ then $\gamma(M) \to_\pi \gamma(N)$*

Proof: By induction on the derivation $M \twoheadrightarrow N$.

R.Fw: $M = \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \twoheadrightarrow \nu k.\, k : Q\{^P/_X\} \mid [a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] = N$. By definition $\gamma(M) = a\langle P\rangle \mid a(X) \triangleright Q \to_\pi Q\{^P/_X\} = \gamma(N)$.

R.Eqv: $M \twoheadrightarrow N$ with hypothesis $M \equiv M'$, $M' \twoheadrightarrow N'$ and $N' \equiv N$. By using the inductive hypothesis we have that $M' \twoheadrightarrow N'$ implies that $\gamma(M') \to_\pi \gamma(N')$ and since structural equivalence is preserved by γ (by Lemma 3.6) we can conclude.

R.Ctx: The empty context case is simply verified by directly applying the inductive hypothesis. For the restriction context we have $\nu u.\, M \twoheadrightarrow \nu u.\, N$ having as hypothesis $M \twoheadrightarrow N$. Now we can distinguish two cases: either $u = a$ (u is a name) or $u = k$ (u is a key). If $u = a$ is a name then by definition $\gamma(\nu a.\, M) = \nu a.\, \gamma(M) \to_\pi \nu a.\, \gamma(N) = \gamma(\nu a.\, N)$ using as hypothesis $\gamma(M) \to_\pi \gamma(N)$ (obtained by applying the inductive hypothesis). If $u = k$ then $\gamma(\nu k.\, M) = \gamma(M)$ and $\gamma(\nu k.\, N) = \gamma(N)$ and we are done by applying the inductive hypothesis. In the case of parallel (right) context we have $M \mid M_1 \twoheadrightarrow N \mid M_1$ having as hypothesis $M \twoheadrightarrow N$. By definition$\gamma(M \mid M_1) = \gamma(M) \mid \gamma(M_1)$ and by applying the inductive hypothesis ($\gamma(M) \to_\pi \gamma(N)$) we have that $\gamma(M) \mid \gamma(M_1) \to_\pi \gamma(N) \mid \gamma(M_1) = \gamma(N \mid M_1)$, as desired.

□

We will now prove the inverse of Lemma 3.7. However γ is not injective, thus it has no inverse. We will show in Lemma 3.10 that for each HOπ process R, each transition $R \to_\pi S$ and each configuration M such that $\gamma(M) = R$ we have a forward reduction in $\rho\pi$ corresponding to $R \to_\pi S$. We start with a few auxiliary results. The first one characterizes the configurations M such that $\gamma(M) = P$ for some HOπ process P.

Lemma 3.8 *Let P be a HOπ process. If $\gamma(M) = P$ and $P \equiv_\pi P'$ with $P' = \nu\tilde{a}.\, \prod_{i\in I} \rho_i$ and $\tilde{a} \subseteq \mathtt{fn}(\prod_{i\in I} \rho_i)$ then $M \equiv \nu\tilde{a}, \tilde{a}', \tilde{k}.\, \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} m_j$ with $\tilde{a}' \cap \mathtt{fn}(\prod_{i\in I} \kappa_i : \rho_i) = \emptyset$.*

Proof: By Lemma 3.1 $M \equiv \nu\tilde{u}.\ \prod_{i' \in I'}(\kappa'_i : \rho'_i) \mid \prod_{j' \in J'} m_{j'} \equiv \nu\tilde{b},\tilde{h}.\ \prod_{i' \in I'}(\kappa'_i : \rho'_i) \mid \prod_{j' \in J'} m_{j'} = M'$ in which we differentiate names from keys. By definition $\gamma(M') = \nu\tilde{b}.\ \prod_{i' \in I'} \rho_{i'}$ but since $M \equiv M'$ by Lemma 3.6 we also have $\gamma(M) \equiv_\pi \gamma(M')$ and since $\gamma(M) = P \equiv_\pi P'$ we have $P' \equiv_\pi \gamma(M')$. Thus we have to prove that if $\nu\tilde{b}.\ \prod_{i' \in I'} \rho_{i'} \equiv_\pi \nu\tilde{a}.\ \prod_{i \in I} \rho_i$ then $\nu\tilde{b},\tilde{h}.\ \prod_{i' \in I'}(\kappa'_i : \rho'_i) \mid \prod_{j' \in J'} m_{j'} \equiv \nu\tilde{a},\tilde{a}',\tilde{k}.\ \prod_{i \in I} \kappa_i : \rho_i \mid \prod_{j \in J} m_j$. We can set $\tilde{b} = \tilde{b}_1, \tilde{b}_2$ where $\tilde{b}_1 \subseteq fn(\prod_{i' \in I'} \rho_{i'})$ and $\tilde{b}_2 \cap fn(\prod_{i' \in I'} \rho_{i'}) = \emptyset$. We have $\nu\tilde{b}_1.\ \prod_{i' \in I'} \rho_{i'} \equiv_\pi \nu\tilde{a}.\ \prod_{i \in I} \rho_i$ which is derived using only α-conversion and axioms E.ParC, E.ParA and E.NilM. Using the same axioms we can derive also $\nu\tilde{b},\tilde{h}.\ \prod_{i' \in I'}(\kappa'_i : \rho'_i) \mid \prod_{j' \in J'} m_{j'} \equiv \nu\tilde{b}_2,\tilde{a},\tilde{h}.\ \prod_{i \in I}(\kappa_i : \rho_i) \mid \prod_{j' \in J'} m'_{j'}$ (memories may be affected by α-conversion). The thesis follows by choosing $\tilde{a}' = \tilde{b}_2$, $\tilde{h} = \tilde{k}$ and $\prod_{j' \in J'} m'_{j'} = \prod_{j \in J} m_j$. □

The next lemma is the inverse of Lemma 3.6.

Lemma 3.9 *If $P \equiv_\pi P'$ then for each M such that $\gamma(M) = P$ there is N such that $N \equiv M$ and $\gamma(N) = P'$.*

Proof: Since both $\equiv_\pi$ and $\equiv$ are equivalence relations, it is enough to show the thesis for derivations of length one. We prove this by structural induction on M.

$M = 0$**:** also $\gamma(M) = 0$. Thus the only axioms that can be applied are E.NilM leading to $0 \mid 0$ and E.NewN leading to $\nu a.\, 0$. The corresponding axioms can be applied to M producing the same terms, which are preserved by γ.

$M = \nu u.\, M_1$**:** here we have to distinguish two cases: either u is a key k or u is a name a. In the first case we have that $\gamma(\nu k.\, M_1) = \gamma(M_1) = P$. We can now apply the inductive hypothesis on $P \equiv_\pi P'$ and we know that there exists an N_1 such that $\gamma(N_1) = P'$ and $M_1 \equiv N_1$, so we also have that $\nu k.\, M_1 \equiv \nu k.\, N_1$ as desired. In the second case we have $P = \gamma(\nu a.\, M_1) = \nu a.\, \gamma(M_1) = \nu a.\, P_1$. If axioms are applied inside P_1 then the thesis follows by inductive hypothesis. The only axioms that can be applied to the whole term are E.NilM, E.NewN, E.NewC, E.NewP and E.α. We will describe in detail the cases for E.NewC and E.NewP, the others are similar.

For E.NewC to apply we need $P_1 = \nu b.\, P_2$ thus $\gamma(M_1) = \nu b.\, P_2$. We will prove the thesis by structural induction on M_1. The only possibilities are: $M_1 = \nu b.\, M_2$, $M_1 = \nu k.\, M_2$ and $M_1 = \kappa : R$ with $R = \nu b.\, R'$. In the first case the thesis follows by using axiom E.NewC on names a

and b. In the second case the thesis follows by applying axiom E.NewC to name b and k and then by applying inductive hypothesis on M_2. For the third case the thesis follows by applying axiom E.TagN to prove that $M \equiv \kappa : \nu a, b.\, R'$ and thus $M \equiv \kappa : \nu b, a.\, R'$ as desired.

For E.NewP to apply we need $P_1 = P_2 \mid P_3$. As before we prove the thesis by structural induction on M_1. The only possibilities are $M_1 = \nu k.\, M_2$, $M_1 = M_2 \mid M_3$ and $M_1 = \kappa : R$ with $R = R_1 \mid R_2$. In the first case the thesis follows by applying axiom E.NewC on names a and k and then by applying inductive hypothesis on M_2. The second case follows by applying axiom E.NewP. Note that if name a occurs in a memory inside M_3 we can use associativity and commutativity of parallel composition to move it inside M_2 and in all the other cases γ does not remove names. In the third case the thesis follows by applying axiom E.TagN to prove that $M \equiv \kappa : \nu a.\, R_1 \mid R_2$ and thus $M \equiv \kappa : (\nu a.\, R_1) \mid R_2$. Note that the side condition is satisfied since γ is the identity on processes (not configurations).

$M = M_1 \mid M_2$: we have $\gamma(M) = \gamma(M_1) \mid \gamma(M_2)$. If axioms are applied inside $\gamma(M_1)$ or inside $\gamma(M_2)$ then the thesis follows by inductive hypothesis. The only axioms that can be applied to the whole parallel composition are: E.ParC, E.ParA, E.NilN and E.α. We will describe in details the cases for E.ParC and E.ParA.

For E.ParC we can apply the same axiom to M and we are done.

For E.ParA to be applicable from left to right (the other direction is symmetric) we need $\gamma(M_2) = R_1 \mid R_2$ thus $M_2 = M_2' \mid M_2''$ with $\gamma(M_2') = R_1$ and $\gamma(M_2'') = R_2$. Thus we can apply the same axiom also to $M_1 \mid (M_2' \mid M_2'')$ and we are done.

$M = \kappa : P$: we have $\gamma(M) = P$. The only possibility is that axioms are applied inside P thus the thesis follows from the definition of γ.

$M = [\mu; k]$: we have $\gamma(M) = 0$, thus the case is similar to the one for $M = 0$.

□

We can finally prove the inverse of Lemma 3.7.

Lemma 3.10 *For all closed HOπ terms R, S if $R \rightarrow_\pi S$ then for all closed configurations M such that $\gamma(M) = R$ there is N such that $M \twoheadrightarrow N$ and $\gamma(N) = S$.*

Proof: By induction on the derivation of the reduction $\rightarrow_\pi$.

Com: $R = a\langle P\rangle \mid a(X) \triangleright Q \rightarrow_\pi Q\{^P/_X\} = S$. Since $\gamma(M) = R$ by Lemma 3.8 we have that $M \equiv \nu \tilde{a'}, \tilde{k}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid M_1$ with $\tilde{a'} \cap \mathtt{fn}(\prod_{i\in I} \kappa_i : \rho_i) = \emptyset$ and M_1 composed only by memories. We have that $M \twoheadrightarrow \nu \tilde{a'}, \tilde{k}, h.\, h : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; h] \mid M_1 = N$. Also, $\gamma(N) = \nu \tilde{a'}.\, Q\{^P/_X\} \equiv_\pi Q\{^P/_X\} = S$ as required.

Eqv: we have that $R \rightarrow_\pi S$ with hypothesis $R \equiv_\pi R'$, $R' \rightarrow_\pi S'$ and $S' \equiv_\pi S$. Taken M such that $\gamma(M) = R$ from Lemma 3.9 there is $M' \equiv M$ such that $\gamma(M') = R'$. Then by inductive hypothesis there is M'' such that $M' \twoheadrightarrow M''$ and $\gamma(M'') = S'$. By applying again Lemma 3.9 we know that there is M''' such that $M''' \equiv M''$ and $\gamma(M''') = S$. The thesis follows by applying rule R.Eqv.

Ctx: we have $\mathbb{C}[R] \rightarrow_\pi \mathbb{C}[S]$ with hypothesis $R \rightarrow_\pi S$. Take M such that $\gamma(M) = \mathbb{C}[R]$. Then there are $\mathbb{C}'$ and M' such that $M = \mathbb{C}'[M']$ and $\gamma(M') = R$. Thus the thesis follows by inductive hypothesis using rule R.Ctx.

□

Remark 3.6 *A canonical way of lifting a closed HOπ process P to a closed consistent configuration in $\rho\pi$ is by defining $\delta(P) = \nu k.\, k : P$. As corollary of Lemma 3.7 we have:*

Corollary 3.1 *For each closed HOπ process P, if $\delta(P) \twoheadrightarrow N$ then $P \rightarrow_\pi \gamma(N)$.*

The Loop Lemma below shows that forward and backward reductions in $\rho\pi$ are really the inverse of each other.

Lemma 3.11 (Loop Lemma) *For all closed consistent configurations M, N if $M \twoheadrightarrow N$ then $N \rightsquigarrow M$, and if $M \rightsquigarrow N$ then $N \twoheadrightarrow M$.*

Proof: Let us start with the first implication. From Lemma 3.4 we have $M \equiv M'$ with $M' = \nu \tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j]$ and $\nu \tilde{u}, k.\, k : Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} [\mu_j; k_j] = N'$ with $N' \equiv N$. Then by applying Lemma 3.5 we have $N \rightsquigarrow M$, as desired.

For the other direction from Lemma 3.5 we have $M \equiv M'$ with $M' = \nu\tilde{u}, k.\, k : R \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j, k_j]$ and $\nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j, k_j] \equiv N$. Since M is a consistent configuration and consistency is preserved by structural congruence also M' is a consistent configuration. Now from consistency properties we know that $k : R$ is the only thread process tagged by k and that $\nu\tilde{u}, k.\, k : R \equiv \nu\tilde{u}, k.\, k : Q\{{}^{P}/{}_{X}\}$. Then the result follows from Lemma 3.4. □

An easy induction on the length n of the sequence of reductions $M \Rightarrow N$ shows that:

Corollary 3.2 *For all closed consistent configurations M, N if $M \Rightarrow N$ then $N \Rightarrow M$.*

3.4 Contextual equivalence in $\rho\pi$

We can classically complement the operational semantics of the $\rho\pi$ calculus with the definition of a contextual equivalence between configurations, which takes the form of a barbed congruence. We first define observables in configurations. We say that name a is *observable in configuration M*, noted $M \downarrow_a$, if $M \equiv \nu\tilde{u}.\, (\kappa : a\langle P\rangle) \mid N$, with $a \notin \tilde{u}$. Note that keys are not observable: this is because they are just an internal device used to support reversibility. We write $M\mathcal{R} \downarrow_a$, where $\mathcal{R}$ is a binary relation on configurations, if there exists N such that $M\mathcal{R}N$ and $N \downarrow_a$. The following definitions are classical:

Definition 3.3 (Barbed bisimulation and congruence) *A relation $\mathcal{R} \subseteq \mathcal{C}^{cl} \times \mathcal{C}^{cl}$ on closed configurations is a* strong *(resp.* weak*)* barbed simulation *if whenever $M\, \mathcal{R}\, N$*

- *$M \downarrow_a$ implies $N \downarrow_a$ (resp. $N \Rightarrow\downarrow_a$)*
- *$M \rightarrow M'$ implies $N \rightarrow N'$, with $M'\mathcal{R}N'$ (resp. $N \Rightarrow N'$ with $M'\mathcal{R}N'$)*

A relation $\mathcal{R} \subseteq \mathcal{C}^{cl} \times \mathcal{C}^{cl}$ is a strong *(resp.* weak*)* barbed bisimulation *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* barbed bisimilarity *and note $\dot{\sim}$ (resp. $\dot{\approx}$) the largest strong (resp. weak) barbed bisimulation. The largest congruence included in $\dot{\sim}$ (resp. $\dot{\approx}$) is called* strong *(resp.* weak*)* barbed congruence *and is noted $\dot{\sim}_c$ (resp. $\dot{\approx}_c$).*

A direct consequence of the Loop Lemma is that each closed consistent configuration M is weakly barbed congruent to any of its descendants or predecessors.

Lemma 3.12 *For all closed consistent configurations M, N, if $M \Rightarrow N$, then $M \dot{\approx}_c N$.*

Proof: We show that the relation

$$\mathcal{R} = \{(\mathbb{C}[M], \mathbb{C}[N]) \mid M \Rightarrow N, \mathbb{C} \text{ is a configuration context}\}$$

is a weak barbed bisimulation. Since $\mathcal{R}$ is symmetric by the corollary of the Loop Lemma (Corollary 3.2), we only need to show that it is a weak barbed simulation. Consider a pair $(\mathbb{C}[M], \mathbb{C}[N]) \in \mathcal{R}$. We have $M \Rightarrow N$, and hence by the Loop Lemma corollary $N \Rightarrow M$. Noting that a configuration context $\mathbb{C}$ is an execution context, i.e. if $M \to N$ then $\mathbb{C}[M] \to \mathbb{C}[N]$, then we also have $\mathbb{C}[N] \Rightarrow \mathbb{C}[M]$. We now check easily the two barbed simulation clauses:

- if $\mathbb{C}[M] \downarrow_a$, then $\mathbb{C}[N] \Rightarrow \mathbb{C}[M] \downarrow_a$, and hence $\mathbb{C}[N] \Rightarrow\downarrow_a$ as required.
- if $\mathbb{C}[M] \to M'$, then $\mathbb{C}[N] \Rightarrow \mathbb{C}[M] \to M'$, and hence $\mathbb{C}[N] \Rightarrow M'$, as required.

Since $\mathcal{R}$ is a congruence by construction the thesis follows. □

Lemma 3.12 shows that barbed bisimilarity is not very discriminative among configurations: $M \dot{\approx} N$ if and only if M and N have the same observables (or N is derived by M). Moreover, we also have that a configuration M is weak barbed bisimilar to a *dummy* configuration that shows directly all the possible barbs of M.

Example 3.1 *Let M and N be defined as follows:*

$$M = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright b\langle X\rangle)$$

$$M_1 = \nu k.\, (k : b\langle 0\rangle) \mid [(k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright b\langle X\rangle); k]$$

$$N = (k_1 : a\langle 0\rangle) \mid (k_2 : b\langle 0\rangle)$$

Then we have that $M \dot{\approx} N$. Let us check it. For barbs, we have that $M \downarrow_a$ and $N \downarrow_a$ and vice versa. For the extra barb on b shown by N we have that $M \to M_1 \downarrow_b$, and barbs are matched. For the reductions, we can note that all the executions of M and M_1 are always matched by a zero execution of N and we are done.

Back and Forth Bisimulations. Other authors already studied the notion of back and forth bisimulations (see [33, 51, 82, 83]) but just considered strong versions in the context of forward calculi. That is, they exploit backward steps as an auxiliary mechanism to better understand purely forward computations. Indeed, this kind of relations can distinguish *true concurrency aspects* better than normal strong bisimulation. For example, the two (CCS) processes:

$$P = a \mid b \qquad Q = a.b + b.a$$

are strongly bisimilar, but using a notion of back and forth bisimulation differ, since if P does a and b, then it can *undo* a, and this computation cannot be matched by Q, because after mimicking the reduction ab it cannot undo a, since b causally depends on a.

Finding a good notion of behavioral equivalence in a reversible calculus is not an easy task. Indeed such an equivalence should be neither trivial nor too strong. An immediate adaptation of the strong barbed congruence leads to a too discriminative relation, while a reversible variant of weak barbed congruence leads to a coarse relation. As shown by the discussion before, adapting the canonical notions [87, 89] of barbed congruence in a reversible setting does not work well, especially for the weak one. Moreover, as shown by Example 3.1, the real problem in the weak equivalence comes from the fact that a process is weakly barbed bisimilar to a dummy one showing all its barbs. So, one may think to consider variants of weak equivalence where forward steps (and barbs) are matched just by forward ones, and backward ones by backward steps (and barbs). In this way, there are several relations that one can define, depending on a specific case. We left to further studies the finding of a good behavioral equivalence for reversible calculi, and in this thesis we will just focus on a particular relation that takes also into account possible administrative steps produced by our encoding.

To prove the *faithfulness* of our encoding (see Section 5.3) with respect to $\rho\pi$, we define an *ad-hoc* notion of behavioral equivalence introducing a new kind of reductions $\hookrightarrow$, namely *administrative* reductions. This is necessary since the encoding introduces reductions that cannot be considered either forward or backward, so the equivalence we are looking for has to work *up-to* administrative reductions. But this does not imply that this relation is the good candidate to become a canonical one on reversible calculi. Indeed if it is used to relate processes of calculi where $\hookrightarrow$ is the empty relation, then this relation becomes stronger than normal strong barbed bisimulation (as we will discuss later on).

We then define a notion of *backward and forward simulation*. We note $\twoheadrightarrow^*$ and $\rightsquigarrow^*$ the reflexive and transitive closure of $\twoheadrightarrow$ and $\rightsquigarrow$, respectively. Moreover we note $\hookrightarrow$ as the reduction relation composed by administrative steps, and $\hookrightarrow^*$ its reflexive and transitive closure. Let us note that in $\rho\pi$ relation $\hookrightarrow$ is left *empty*. We now characterize a new form of barbed bisimulation taking into account also administrative steps.

Definition 3.4 (Backward-and-forward barbed bisimulation and congruence) *A relation $\mathcal{R} \subseteq \mathcal{C}^{cl} \times \mathcal{C}^{cl}$ on closed configurations is a* strong *(resp.* weak*)* backward-and-forward barbed simulation *(or* bf barbed simulation *for brevity) if whenever $M \,\mathcal{R}\, N$*

- *$M \downarrow_a$ implies $N \downarrow_a$ (resp. $N \hookrightarrow^* \downarrow_a$)*
- *$M \twoheadrightarrow M'$ implies $N \twoheadrightarrow N'$, with $M'\mathcal{R}N'$ (resp. $N \hookrightarrow^* \twoheadrightarrow \hookrightarrow^* N'$ with $M'\mathcal{R}N'$)*
- *$M \rightsquigarrow M'$ implies $N \rightsquigarrow N'$, with $M'\mathcal{R}N'$ (resp. $N \hookrightarrow^* \rightsquigarrow \hookrightarrow^* N'$ with $M'\mathcal{R}N'$)*
- *$M \hookrightarrow M'$ implies $N \hookrightarrow N'$, with $M'\mathcal{R}N'$ (resp. $N \hookrightarrow^* N'$ with $M'\mathcal{R}N'$)*

A relation $\mathcal{R} \subseteq \mathcal{C}^{cl} \times \mathcal{C}^{cl}$ is a strong *(resp.* weak*)* barbed bisimulation *if $\mathcal{R}$ and $\mathcal{R}^{-1}$ are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* back-and-forth barbed bisimilarity *and note $\overset{\circ}{\sim}$ (resp. $\overset{\circ}{\approx}$) the largest strong (resp. weak) barbed bisimulation. The largest congruence included in $\overset{\circ}{\sim}$ (resp. $\overset{\circ}{\approx}$) is called* strong *(resp.* weak*)* barbed congruence *and is noted $\overset{\circ}{\sim}_c$ (resp. $\overset{\circ}{\approx}_c$).*

Weak bf barbed bisimulation is just weak with respect to administrative reductions $\hookrightarrow$. If it is used to equate $\rho\pi$ processes, we have that the strong version and the weak one coincide, that is $\overset{\circ}{\approx} = \overset{\circ}{\sim}$. Hence, in $\rho\pi$, weak bf barbed bisimulation becomes a stronger relation than (normal) strong barbed bisimulation, since it distinguishes the direction of reductions and a forward (backward) step is matched by just one forward (backward) step. For example:

$$A = \nu k_1, k_2.\, (k_1 : a\langle \mathbf{0} \rangle) \mid (k_2 : a(X) \triangleright a\langle \mathbf{0} \rangle \mid b\langle P \rangle \mid P)$$

$$P = b(Y) \triangleright a(X) \triangleright Y \mid b\langle Y \rangle \mid a\langle \mathbf{0} \rangle$$

$$B = \nu k_1, k_2.\, (k_1 : a\langle \mathbf{0} \rangle) \mid (k_2 : a(X) \triangleright a\langle \mathbf{0} \rangle \mid b\langle \mathbf{0} \rangle)$$

We have $A \mathrel{\dot{\sim}} B$ but $A \mathrel{\overset{\circ}{\not\sim}} B$. Moreover, if we take M and N of the Example 3.1 we have that $M \mathrel{\overset{\circ}{\not\approx}} N$ while $M \mathrel{\dot{\approx}} N$.

From the definitions, it is clear that $\overset{\circ}{\sim}\subseteq\dot{\sim}$, $\overset{\circ}{\approx}\subseteq\dot{\approx}$, $\overset{\circ}{\sim}_c\subseteq\dot{\sim}_c$, and $\overset{\circ}{\approx}_c\subseteq\dot{\approx}_c$. All these inclusions are strict, however. For instance, consider the following configurations:

$$\begin{aligned}
M &= \nu k_1, k_2, k_3.\, (k_1 : a\langle \mathbf{0} \rangle) \mid (k_2 : a(X) \triangleright b\langle \mathbf{0} \rangle) \mid (k_3 : a(X) \triangleright c\langle \mathbf{0} \rangle)\\
N &= \nu k_1, k_2, k_3, k_4.\, (k_4 : b\langle \mathbf{0} \rangle) \mid (k_3 : a(X) \triangleright c\langle \mathbf{0} \rangle) \mid\\
&\qquad [(k_1 : a\langle \mathbf{0} \rangle) \mid (k_2 : a(X) \triangleright b\langle \mathbf{0} \rangle); k_4]\\
M' &= \nu k_1, k_2, k_3, k_5.\, (k_5 : c\langle \mathbf{0} \rangle) \mid (k_2 : a(X) \triangleright b\langle \mathbf{0} \rangle) \mid\\
&\qquad [(k_1 : a\langle \mathbf{0} \rangle) \mid (k_3 : a(X) \triangleright c\langle \mathbf{0} \rangle); k_5]
\end{aligned}$$

We have $M \twoheadrightarrow N$, $M \mathrel{\dot{\approx}_c} N$, but $M \mathrel{\overset{\circ}{\not\approx}} N$ and hence $M \mathrel{\overset{\circ}{\not\approx}_c} N$. To prove this, assume that there exists a weak bf barbed bisimulation $\mathcal{R}$ between M and N, i.e. such that $(M, N) \in \mathcal{R}$. We have $M \twoheadrightarrow M'$, and we have no N' such that $N \twoheadrightarrow N'$, hence we must have $(M', N) \in \mathcal{R}$. Now, $M' \downarrow_c$ but we do not have $N \downarrow_c$, nor $N \hookrightarrow^* \downarrow_c$ since we only have $N \rightsquigarrow M$, and $M \not\downarrow_c$, which contradicts $(M', N) \in \mathcal{R}$.

3.5 Causality

We now proceed to the analysis of causality in $\rho\pi$, showing that reversibility in $\rho\pi$ is causally consistent, that is during backward steps we may end up in a state that we could have reached during the forward computation by just swapping the execution order of concurrent actions. We mostly adapt for the exposition the terminology and arguments of RCCS (see [29]).

We call *transition* a triplet of the form $M \xrightarrow{m_{\twoheadrightarrow}} M'$, or $M \xrightarrow{m_{\rightsquigarrow}} M'$, where M, M' are closed consistent configurations, $M \to M'$, and m is the memory involved in the reduction $M \to M'$. We indicate M as the *source* of the transition and M' as the *target* of the transition. We say that a memory m is *involved* in a reduction $M \twoheadrightarrow M'$ if $M \equiv \mathbb{E}[\kappa_1 : a\langle P \rangle \mid \kappa_2 : a(X) \triangleright Q]$, $M' \equiv \mathbb{E}[\nu k.\, (k : Q\{^P/_X\}) \mid m]$, and $m = [\kappa_1 : a\langle P \rangle \mid \kappa_2 : a(X) \triangleright Q; k]$. In this case, the transition involving m is noted $M \xrightarrow{m_{\twoheadrightarrow}} M'$. Likewise, we say that a memory $m = [N; k]$ is involved in a reduction $M \rightsquigarrow M'$ if $M \equiv \mathbb{E}[(k : Q) \mid m]$, $M' \equiv \mathbb{E}[N]$. In this case, the transition involving m is noted $M \xrightarrow{m_{\rightsquigarrow}} M'$. In a transition $M \xrightarrow{\eta} N$, we say that M is the *source* of the transition, N is its *target*, and η is its label (of the form $m_{\twoheadrightarrow}$ or $m_{\rightsquigarrow}$, where m is some memory – we let η and its decorated variants range over transition labels). If $\eta = m_{\twoheadrightarrow}$, we set $\eta_\bullet = m_{\rightsquigarrow}$ and vice versa.

Definition 3.5 (Name-preserving transitions) *We say a transition $t : M \xrightarrow{\eta} M'$ is* name-preserving *if M and M' are in thread normal form and if one of the following assumptions holds:*

1. *$M = \nu\tilde{u}. \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[M_j : k_j]$, $M' = \nu\tilde{u}. \prod_{i\in I'}(\kappa_i : \rho_i) \mid \prod_{j\in J'}[M_j : k_j]$, with $J' = J \cup \{j'\}$, $I' \subset I$ and $\eta = m_{\twoheadrightarrow}$ with $m = [M_{j'}; k_{j'}]$;*

2. *$M = \nu\tilde{u}. \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[M_j : k_j]$, $M' = \nu\tilde{u}. \prod_{i\in I'}(\kappa_i : \rho_i) \mid \prod_{j\in J'}[M_j : k_j]$, with $J = J' \cup \{j'\}$, $I \subset I'$ and $\eta = m_{\rightsquigarrow}$ with $m = [M_{j'}; k_{j'}]$.*

Intuitively, a name-preserving transition keeps track of its restricted names: all the names used in the transition (and especially the tag of memory m) are preserved by the transition, said otherwise names of tags are never α-converted.

Remark 3.7 *In the rest of this section we only consider* name-preserving *transitions and "transition" used in a definition, lemma or theorem, stands for "name-preserving transition". Note that working with name-preserving transitions only is licit because of the determinacy lemma (Lemma 3.2).*

Two transitions are said to be *coinitial* if they have the same source, *cofinal* if they have the same target, and *composable* if the target of one is the source of the other. A sequence of pairwise composable transitions is called a *trace*. We let t and its decorated variants range over transitions, σ and its decorated variants range over traces. Notions of target, source and composability extend naturally to traces. We denote with ϵ_M the empty trace with source M, $\sigma_1; \sigma_2$ the composition of two composable traces σ_1 and σ_2. We now define the stamp of a memory m, a notion that will be useful to know when two transitions are concurrent.

Definition 3.6 (Memory Stamp) *The* stamp *$\lambda(m)$ of a memory $m = [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k]$ is defined as:*

$$\lambda(m) = \{\kappa_1, \kappa_2, k\}$$

we set $\lambda(m_{\twoheadrightarrow}) = \lambda(m_{\rightsquigarrow}) = \lambda(m)$.

Definition 3.7 (Concurrent transitions) *Two coinitial transitions $t_1 = M \xrightarrow{\eta_1} M_1$ and $t_2 = M \xrightarrow{\eta_2} M_2$ are said to be* concurrent *if $\lambda(\eta_1) \cap \lambda(\eta_2) = \emptyset$.*

$$\begin{tikzcd}
& M \arrow[dl, two heads, "{[\mu_1,k_5]}"'] \arrow[dr, two heads, "{[\mu_2,k_6]}"] & \\
M_1 & & M_2
\end{tikzcd}$$

Figure 3.4: Example concurrent transitions.

$$\begin{tikzcd}
& M \arrow[dl, two heads, "{[\mu_1,k_5]}"'] \arrow[dr, two heads, "{[\mu_2,k_6]}"] & \\
M_1 \arrow[dr, two heads, "{[\mu_2,k_6]}"'] & & M_2 \arrow[dl, two heads, "{[\mu_1,k_5]}"] \\
& N &
\end{tikzcd}$$

Figure 3.5: Example of Square Lemma.

An example of concurrent transitions is given in Figure 3.4 where:

$$\begin{array}{ll}
M = \mu_1 \mid \mu_2 & M_1 = \nu k_5.\, k_5 : 0 \mid [\mu_1; k_5] \mid \mu_2 \\
M_2 = \nu k_6.\, k_6 : 0 \mid [\mu_2; k_6] \mid \mu_1 & \mu_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright 0) \\
\mu_2 = (k_3 : b\langle 0\rangle) \mid (k_4 : b(X) \triangleright 0) & \eta_1 = [\mu_1; k_5] \\
\eta_2 = [\mu_2; k_6] &
\end{array}$$

We have that the transitions $t_1 : M \xrightarrow{\eta_1 \twoheadrightarrow} M_1$ and $t_2 : M \xrightarrow{\eta_2 \twoheadrightarrow} M_2$ are concurrent since $\lambda(\eta_1) = \{k_1, k_2, k_5\}$, $\lambda(\eta_2) = \{k_3, k_4, k_6\}$ and $\{k_1, k_2, k_5\} \cap \{k_3, k_4, k_6\} = \emptyset$.

Remark 3.8 *Note that the stamp of a memory $[\mu; k]$ includes its tag k. This is necessary to take into account possible conflicts between a forward action and a backward action. Here is an example: the configuration*

$$M = \nu l, k, h.\, (k : a\langle P\rangle) \mid [N; k] \mid (h : a(X) \triangleright Q)$$

has two possible transitions $t = M \xrightarrow{m \rightsquigarrow} \nu l, k, h.\, N \mid (h : a(X) \triangleright Q)$, where $m = [N; k]$, and $t' = M \xrightarrow{m' \twoheadrightarrow} \nu l, k, h.\, [N; k] \mid m' \mid l : Q\{^P/_X\}$, where $m' = [(k : a\langle P\rangle) \mid (h : a(X) \triangleright Q); l]$. The two transitions t and t' are in conflict over the use of the resource $k : a\langle P\rangle$.

The Loop Lemma ensures that each transition $t = M \xrightarrow{\eta} N$ has a reverse one $t_\bullet = N \xrightarrow{\eta_\bullet} M$.

The following lemma (Square Lemma) is a crucial lemma for our proof strategy. It states that in the presence of two concurrent transitions, the order in which they are executed does not matter. In this way, when dealing with a sequence of transitions we can change the order (execution) of two concurrent transitions, obtaining the same result in terms of reached configurations. The above definition of concurrent transitions makes sense:

Lemma 3.13 (Square Lemma) *If $t_1 = M \xrightarrow{\eta_1} M_1$ and $t_2 = M \xrightarrow{\eta_2} M_2$ are two coinitial concurrent transitions, then there exist two cofinal transitions $t_2/t_1 = M_1 \xrightarrow{\eta_2} N$ and $t_1/t_2 = M_2 \xrightarrow{\eta_1} N$.*

Proof: By case analysis on the form of transitions t_1 and t_2. We have that from M there are two possible concurrent transitions t_1 and t_2. We proceed by case analysis on the possible combinations of $\rightarrow$.

- $M \xrightarrow{m_1 \twoheadrightarrow} N_1$ and $M \xrightarrow{m_2 \twoheadrightarrow} N_2$. By Lemma 3.4 if $M \twoheadrightarrow N_1$ then $M \equiv M'$ with:

  $$M' = \nu\tilde{u}.\,(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \mid \prod_{i \in I} \kappa_i : \rho_i \mid \prod_{j \in J} [\mu_j ; k_j]$$
  $$N' = \nu\tilde{u}, k.\,(k : Q\{^{P}/_{X}\}) \mid m_1 \mid \prod_{i \in I} \kappa_i : \rho_i \mid \prod_{j \in J} [\mu_j ; k_j]$$

  and $m_1 = [\kappa_1 : a\langle P\rangle \mid a(X) \triangleright Q; k]$, $N' \equiv N_1$. For the same reason if $M \twoheadrightarrow N_2$ then $M \equiv M''$ with:

  $$M'' = \nu\tilde{u}.\,(\kappa'_1 : a'\langle P'\rangle) \mid (\kappa'_2 : a'(X) \triangleright Q') \mid \prod_{i \in I'} \kappa_i : \rho_i \mid \prod_{j \in J} [\mu_j ; k_j]$$
  $$N' = \nu\tilde{u}, k'.\, k' : Q'\{^{P'}/_{X}\} \mid m_2 \mid \prod_{i \in I'} \kappa_i : \rho_i \mid \prod_{j \in J} [\mu_j ; k_j]$$

  and with $m_2 = [\kappa'_1 : a'\langle P'\rangle \mid a'(X) \triangleright Q'; k']$, $N'' \equiv N_2$. Since the two transitions are concurrent (by hypothesis) we have that $\{\kappa_1, \kappa_2, k\} \cap \{\kappa'_1, \kappa'_2, k'\} = \emptyset$. Thus, we have:

  $$M \equiv \nu\tilde{u}.\,(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \mid (\kappa'_1 : a'\langle P'\rangle) \mid (\kappa'_2 : a'(X) \triangleright Q') \mid \prod_{i \in I''} \kappa_i : \rho_i \mid \prod_{j \in J} m_j$$
  $$N_1 \equiv \nu\tilde{u}, k.\,(\kappa'_1 : a'\langle P'\rangle) \mid (\kappa'_2 : a'(X) \triangleright Q') \mid \prod_{i \in I''} \kappa_i : \rho_i \mid \prod_{j \in J} m_j \mid m_1$$

with $m_1 = [\kappa_1 : a\langle P\rangle \mid a(X) \triangleright Q; k]$ and

$$N_2 \equiv \nu\tilde{u}, k'.\, (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q) \mid \prod_{i\in I''} \kappa_i : \rho_i \mid (\prod_{j\in J} m_j) \mid m_2$$

We have that $N_2 \xrightarrow{m_1 \twoheadrightarrow} \nu\tilde{u}, k', k.\, \prod_{i\in I''} \kappa_i : \rho_i \mid (\prod_{j\in J} m_j) \mid m_2 \mid m_1$ and $N_1 \xrightarrow{m_2 \twoheadrightarrow} \nu\tilde{u}, k', k.\, \prod_{i\in I''} \kappa_i : \rho_i \mid (\prod_{j\in J} m_j) \mid m_2 \mid m_1$, as desired.

- $M \xrightarrow{m_1 \rightsquigarrow} N_1$ and $M \xrightarrow{m_2 \twoheadrightarrow} N_2$. Since $M \xrightarrow{m_1 \rightsquigarrow} N_1$ by Lemma 3.5 $M \equiv M'$ with:

$$M' = \nu\tilde{u}, k.\, k : R \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid \prod_{i\in I'} \kappa_i : \rho_i \mid \prod_{j\in J'} m_j$$
$$N_1 \equiv \nu\tilde{u}.\, \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J} m_j$$

and since $M \xrightarrow{m_2 \twoheadrightarrow} N_2$ then by Lemma 3.4 $M \equiv M''$ with:

$$M'' = \nu\tilde{u}.\, \kappa_1' : a'\langle P'\rangle \mid \kappa_2' : a'(X) \triangleright Q' \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J'} m_j$$
$$N'' = \nu\tilde{u}, k'.\, k' : Q'\{^{P'}/_X\} \mid [\kappa_1' : a\langle P'\rangle \mid \kappa_2' : a(X) \triangleright Q'; k'] \mid \prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J'} [\mu_j; k_j]$$

and $N'' \equiv N_2$. By hypothesis the two transitions are concurrent so $\{\kappa_1, \kappa_2, k\} \cap \{\kappa_1', \kappa_2', k'\} = \emptyset$. Thus, we have that

$$M \equiv \nu\tilde{u}, k'.\, k : R \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid (\kappa_1' : a'\langle P'\rangle) \mid (\kappa_2' : a'(X) \triangleright Q') \mid \prod_{i\in I''} \kappa_i : \rho_i \mid \prod_{j\in J''} m_j$$
$$N_1 \equiv \nu\tilde{u}.\, \kappa_1' : a'\langle P\rangle \mid \kappa_2' : a'(X) \triangleright Q' \mid \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \mid \prod_{i\in I''} \kappa_i : \rho_i \mid \prod_{j\in J''} m_j$$
$$N_2 \equiv \nu\tilde{u}, k', k.\, [\kappa_1' : a'\langle P\rangle \mid \kappa_2' : a'(X) \triangleright Q'; k'] \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \mid k' : Q'\{^{P'}/_X\} \mid k : R \mid \prod_{i\in I''} \kappa_i : \rho_i \mid \prod_{j\in J''} m_j$$

We have that:

$$\begin{aligned}
N_2 &\xrightarrow{m_1 \rightsquigarrow} \nu \tilde{u}, k'.\, (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q) \mid [\kappa'_1 : a'\langle P' \rangle \mid \kappa'_2 : a'(X) \triangleright Q'; k'] \mid \\
&\qquad k' : Q'\{^{P'}/_X\} \mid \prod_{i \in I''} \kappa_i : \rho_i \mid \prod_{j \in J''} m_j \\
N_1 &\xrightarrow{m_2 \twoheadrightarrow} \nu \tilde{u}, k'.\, (\kappa_1 : a\langle P \rangle) \mid (\kappa_2 : a(X) \triangleright Q) \mid [\kappa'_1 : a\langle P' \rangle \mid \kappa'_2 : a(X) \triangleright Q'; k'] \mid \\
&\qquad k' : Q'\{^{P'}/_X\} \mid \prod_{i \in I''} \kappa_i : \rho_i \mid \prod_{j \in J''} m_j
\end{aligned}$$

as desired.

- $M \xrightarrow{m_1 \twoheadrightarrow} N_1$ and $M \xrightarrow{m_2 \rightsquigarrow} N_2$, similar to the case above.
- $M \xrightarrow{m_1 \rightsquigarrow} N_1$ and $M \xrightarrow{m_2 \rightsquigarrow} N_2$, similar to the first case.

□

Continuing the example of Figure 3.4, Figure 3.5 shows graphically how the Square Lemma works.

We are now in a position to show that reversibility in $\rho\pi$ is causally consistent. We define first the notion of causal equivalence between traces, noted $\asymp$.

Definition 3.8 (causal equivalence) *We define $\asymp$ as the least equivalence relation between traces closed under composition that obeys the following rules (where t is forward):*

$$t_1; t_2/t_1 \asymp t_2; t_1/t_2 \qquad\qquad t; t_\bullet \asymp \epsilon_{\texttt{source}(t)} \qquad\qquad t_\bullet; t \asymp \epsilon_{\texttt{target}(t)}$$

The relation $\asymp$ states that if we have two concurrent transitions, then the two traces obtained by swapping the order of their execution are the same, and that executing a reduction followed by its complementary one is equivalent to *doing nothing*, that is the empty trace.

The proof of causal consistency proceeds along the exact same lines as in [29], with simpler arguments because of the simpler form of our memory stamps. We denote as $\sigma_\bullet$ the trace $t_{n\bullet}; \ldots; t_{1\bullet}$ if $\sigma = t_1; \ldots; t_n$.

The following Lemma says that each generic trace σ is causally equivalent to a trace made of a sequence of backward transitions followed by a sequence of forward transitions.

Lemma 3.14 (Rearranging lemma) *Let σ be a trace. There exist forward traces σ' and σ'' such that $\sigma \asymp \sigma'_\bullet; \sigma''$.*

Proof: The proof is by lexicographic induction on the length of σ and on the distance between the beginning of σ and the earliest pair of transitions in σ of the form $t; t'_\bullet$ (where t and t' are forward). If there is no such pair we are done. If there is one, we have two possibilities: either $\lambda(m) \cap \lambda(m') = \emptyset$ or $\lambda(m) \cap \lambda(m') \neq \emptyset$ with m being the memory associated to t and m' the one associated to $t'_\bullet$. In the first case, it means that the two transitions are concurrent and so we can swap them by using Lemma 3.13, resulting in a later earliest contradicting pair, and by induction the result follows since swapping transitions keeps the total length constant. In the second case we have that $\{\delta_1, \gamma_1, k_1\} \cap \{\delta_2, \gamma_2, k_2\} \neq \emptyset$. Thanks to consistency property on configurations the only possibilities are (1) the two memories coincide, or (2) $k_1 = \delta_2$ or $k_1 = \gamma_2$ or (3) $k_2 = \delta_1$ or $k_2 = \gamma_1$. In the first case we have $t = t'$, and we can apply the Loop lemma removing $t; t_\bullet$. Hence the total length of σ decreases and again by induction the result follows. In the second case t has created a memory of the form $[\delta_1 : a\langle P\rangle \mid \gamma_1 : a(X) \triangleright Q; k_1]$ and a process $k_1 : R$. Thus from condition 3 (of consistent configuration) this case never happens. In the last case transition $t'_\bullet$ deletes a memory of the form $[\delta_2 : a\langle P\rangle \mid \gamma_2 : a(X) \triangleright Q; k_2]$, but this requires having a process $k_2 : R$. Again from condition 3 this case never happens. □

The next Lemma states that if there are two traces σ_1, σ_2 that start from the same configuration and end up in the same configuration, with σ_2 made of forward computations, then there exists a trace σ'_1 causally equivalent to σ_1 made of forward computations. In other words, since σ_2 is a forward trace and σ_1, σ_2 are cofinal and coinitial, then all the backward reductions done in the trace σ_1 are useless and can be eliminated.

Lemma 3.15 (Shortening lemma) *Let σ_1, σ_2 be coinitial and cofinal traces, with σ_2 forward. Then, there exists a forward trace σ'_1 of length at most that of σ_1 such that $\sigma'_1 \asymp \sigma_1$.*

Proof: We prove this lemma by induction on the length of σ_1. If σ_1 is a forward trace we are already done.

Otherwise by Lemma 3.14 we can write σ_1 as $\sigma_\bullet; \sigma'$ (with σ and σ' forward). Let $t_\bullet; t'$ be the only two successive transitions in σ_1 with opposite direction, with m_1 belonging to $t_\bullet$. Since m_1 is removed by $t_\bullet$ then m_1 has to be put back by another forward transition otherwise this difference will stay visible since σ_2 is a forward trace. Let t_1 be the earliest such transition in σ_1, since it is able to put back m_1 it has to be the exact opposite of $t_\bullet$, so $t_1 = t$. Now we can swap t_1 with all the transitions between t_1 and $t_\bullet$, in order to obtain a trace in which t_1 and $t_\bullet$ are adjacent. To do so we use the square lemma (Lemma 3.13), since all the transitions in between will are concurrent. Assume in fact that there is a transition involving memory m_2 which is not concurrent to t_1, with $\lambda(m_1) = \{\delta_1, \gamma_1, k_1\}$, $\lambda(m_2) = \{\delta_2, \gamma_2, k_2\}$ and $\{\delta_1, \gamma_1, k_1\} \cap \{\delta_2, \gamma_2, k_2\} \neq \emptyset$. Thanks to consistency conditions the only possibilities are (1) $k_1 = \delta_2$ or $k_1 = \gamma_2$ or (2) $k_2 = \delta_1$ or $k_2 = \gamma_1$. The first case can never happen since k_1 is fresh (generated by the forward rule) and thus cannot coincide with γ_2 or δ_2. Similarly the second case can never happen since k_2 is fresh and thus cannot occur in m_1. When $t_\bullet$ and t are adjacent we can remove both of them using $\asymp$. The resulting trace is shorter, thus the thesis follows by inductive hypothesis. □

$$\begin{tikzcd}
 & M \arrow[dl, two heads, "{[\mu_1,k_6]}"'] \arrow[dr, two heads, "{[\mu_2;k_7]}"] & & & M \arrow[d, two heads, "{[\mu_2;k_7]}"] \\
M_1 \arrow[dr, two heads, "{[\mu_2;k_7]}"'] & & M_2 \arrow[dl, two heads, "{[\mu_1;k_6]}"] & & N_1 \arrow[d, two heads, "{[\mu_3;k_8]}"] \\
 & M_3 \arrow[d, two heads, "{[\mu_3;k_8]}"] & & & N \\
 & M_4 \arrow[d, rightsquigarrow, "{[\mu_1,k_6]}"] & & & \\
 & N & & &
\end{tikzcd}$$

Figure 3.6: Causally equivalent traces.

Figure 3.6 depicts how a trace can be shortened by using the Shortening Lemma. We have that the two traces from M to N that can be derived from the left part of the Figure 3.6, can be shortened in the trace that can be derived from the right part of the figure, hence the left traces are causally equivalent to the right trace. Configurations used by Figure 3.6 are:

$$M = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X)\triangleright 0) \mid (k_3 : b\langle 0\rangle) \mid (k_4 : b(X)\triangleright c\langle 0\rangle) \mid (k_5 : c(Z)\triangleright 0)$$

$$M_1 = \nu k_6.\, k_6 : 0 \mid [\mu_1; k_6] \mid \mu_2 \mid (k_5 : c(Z) \triangleright 0)$$

$$M_2 = \nu k_7.\, k_7 : c\langle 0\rangle \mid [\mu_2; k_7] \mid \mu_1 \mid (k_5 : c(Z) \triangleright 0)$$

$$M_3 = \nu k_6, k_7.\, k_6 : 0 \mid k_7 : c\langle 0\rangle \mid [\mu_2; k_7] \mid [\mu_1; k_6] \mid (k_5 : c(Z) \triangleright 0)$$

$$M_4 = \nu k_6, k_7, k_8.\, k_6 : 0 \mid [\mu_2; k_7] \mid [\mu_1; k_6] \mid k_8 : 0 \mid [\mu_3; k_8]$$

$$N = \nu k_7, k_8. k_7 : c\langle 0\rangle \mid [\mu_2; k_7] \mid \mu_1 \mid k_8 : 0 \mid [\mu_3; k_8]$$

$$\mu_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright 0) \qquad \mu_2 = (k_3 : b\langle 0\rangle) \mid (k_4 : b(X) \triangleright c\langle 0\rangle)$$
$$\mu_3 = (k_7 : c\langle 0\rangle) \mid (k_5 : c(X) \triangleright 0)$$

We can now state our main Theorem, showing that reductions in $\rho\pi$ are causally consistent.

Theorem 3.1 (Causal consistency) *Let σ_1 and σ_2 be coinitial traces, then $\sigma_1 \asymp \sigma_2$ if and only if σ_1 and σ_2 are cofinal.*

Proof: By construction of $\asymp$, if $\sigma_1 \asymp \sigma_2$ then σ_1 and σ_2 must be coinitial and cofinal, so this direction of the theorem is verified. Now we have to consider that σ_1 and σ_2 being coinitial and cofinal implies that $\sigma_1 \asymp \sigma_2$. By Lemma 3.14 we know that the two traces can be written as composition of a backward trace and a forward one. Be t_1 and t_2 the first two transitions on which the two traces disagree. The proof is by lexicographic induction on the sum of the lengths of σ_1 and σ_2 and on the distance between the end of σ_1 and the earliest pair of transitions in σ_1 and σ_2 which are not equal. If all the transitions are equal then we are done. Otherwise we have to consider three cases depending on the direction of the two transitions.

- If t_1 is forward and t_2 is backward, we have that $\sigma_1 = \sigma_\bullet; t_1; \sigma'$ and $\sigma_2 = \sigma_\bullet; t_2; \sigma''$. Moreover we know that $t_1; \sigma'$ is a forward trace, so we can apply the Lemma 3.15 on the traces $t_1; \sigma'$ and $t_2; \sigma''$ (since σ_1 and σ_2 are coinitial and cofinal by hypothesis, we also have $t_1; \sigma'$ and $t_2; \sigma''$ coinitial and cofinal) and we obtain that $t_2; \sigma''$ has a shorter equivalent forward trace and so also σ_2 has a shorter equivalent forward trace and we can conclude by induction.
- Consider the case both t_1 and t_2 are forward. By assumption the two transitions are different. If they are not concurrent then they should conflict on a thread process $\kappa : P$, that they both consume and store in different memories. Since the two traces are cofinal there should be t_2' in σ_2 creating the same memory as t_1. However no other process $\kappa : P$ is ever created in σ_2 thus this is not possible. So we can assume that t_1 and t_2 are concurrent. Again let t_2' be the transition in σ_2 creating the same memory of t_1. We have to prove that t_2' is concurrent to all the previous transitions. This holds since no previous transition can remove one of the processes needed for triggering t_2' and since forward transitions can never conflict on k. Thus we can repetitively apply the

Square lemma to derive a trace equivalent to σ_2 where t_2 and t'_2 are consecutive. We can apply a similar transformation to σ_1. Now we can apply the Square lemma to t_1 and t_2 to have two traces of the same length as before but where the first pair of different transitions is closer to the end. The thesis follows by inductive hypothesis.

- If both t_1 and t_2 are backward, then they cannot remove the same memory. Let m_1 be the memory removed by t_1. Since the two traces are cofinal, either there is another transition in σ_1 putting back the memory or there is a transition t'_1 in σ_2 removing the same memory. In the first case, t_1 is concurrent to all the backward transitions following it, but the ones that consume processes generated by it. All the transitions of this kind have to be undone by corresponding forward transitions (since they are not possible in σ_2). Consider the last such transition: we can use the Square lemma to make it the last backward transition. The forward transition undoing it should be concurrent to all the previous forward transitions (the reason is the same as in the previous case). Thus we can use the Square lemma to make it the first forward transition. Finally we can apply the Loop lemma to remove the two transitions, thus shortening the trace. The thesis follows by inductive hypothesis.

□

Chapter 4

The **roll**-π calculus

We present in this chapter the study of a fine-grained rollback control primitive, where potentially every single step in a concurrent execution can be undone. Specifically, we introduce a rollback construct for an asynchronous Higher-Order π-calculus (HOπ [86]), exploiting the reversible machinery of $\rho\pi$, the reversible Higher-Order π-calculus (see Chapter 3). Controlling reversibility may seem just a straightforward exercise, since we build this primitive on the top of a reversible calculus [58]. Surprisingly, finding a suitable definition for a fine-grained rollback construct in HOπ is more difficult than one may think. There are two main difficulties. The first one is in actually pinning down the intended effect of a rollback operation, especially in presence of concurrent rollbacks. The second one is in finding a suitably distributed semantics for rollback, dealing only with local information and not relying on complex atomic operations involving potentially an unbounded number of processes.

We show in this chapter how to deal with these difficulties by making the following contributions: (i) we define a high-level operational semantics for a rollback construct in an asynchronous Higher-Order π-calculus, which we prove *maximally permissive*, in the sense that it makes reachable all past states in a given computation; (ii) we present a low-level semantics for the proposed rollback construct which can be understood as a fully distributed variant of our high-level semantics, and we prove it to be *fully abstract* with respect to the high-level one. To prove this, we will refine the high level semantics through several semantics, each time trying to use fewer global checks and atomic steps on whole configurations than before. When we introduce a new refinement semantics, we prove that the refining one is *equivalent* (in terms of weak barbed congruence) to the refined one.

The rest of the chapter is structured as follows: first we informally introduce our rollback primitive and we argue about problems that concurrent rollbacks can cause, then we introduce our calculus, called $\mathsf{roll}\text{-}\pi$ and its high level semantics. We prove that the calculus provides a controlled version of $\rho\pi$ backward reductions. Then we introduce what we call intermediate semantics: several refinements of the high level semantics toward a lower level one. Eventually a low level semantics, close to an actual implementation, of $\mathsf{roll}\text{-}\pi$ is given.

4.1 Informal Presentation

4.1.1 Reversibility in $\mathsf{roll}\text{-}\pi$

The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. In $\mathsf{roll}\text{-}\pi$, we exploit this intuition to introduce an explicit form of backward reduction. Specifically, backward reduction is not allowed by default as in $\rho\pi$, but has to be triggered by an instruction of the form $\mathsf{roll}\ k$, whose intent is that the current computation be rolled-back to a state just prior to the creation of the memory bearing the tag k. To be able to form an instruction of the form $\mathsf{roll}\ k$, one needs a way to pass the knowledge of a memory tag to a process. This is achieved in $\mathsf{roll}\text{-}\pi$ by adding a bound variable to each trigger process, which now takes the form $a(X) \triangleright_\gamma P$, where γ is the tag variable bound by the trigger construct and whose scope is P. A forward reduction step in $\mathsf{roll}\text{-}\pi$ therefore is:

$$(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \quad \twoheadrightarrow \quad \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [M; k] \tag{4.1}$$

where the only difference with $\rho\pi$ forward rule (Figure 3.3 in Section 3.2.1) lies in the fact that the newly created tag k is passed as an argument to the trigger body Q. In this way, all the free occurrences of the process $\mathsf{roll}\ \gamma$ in Q, after the communication, will point to the memory bearing the tag k. We write $a(X) \triangleright P$ in place of $a(X) \triangleright_\gamma P$ if the tag variable γ does not appear free in P.

Given a $\rho\pi$ configuration M, the set of memories present in M provides us with an ordering $:>$ between tags in M that reflects their causal dependency: if memory $[\kappa_1 : P_1 \mid \kappa_2 : P_2; k]$ occurs in M, then $\kappa_i > k$. Also, $k > \langle h_i, \tilde{h}\rangle \cdot k$ (where $\langle h_i, \tilde{h}\rangle \cdot k$ has been derived using the structural law E.TagP of Figure 3.2), and we define the relation $:>$ as the reflexive and transitive closure of the $>$ relation. We say that tag κ has κ' as a causal antecedent if $\kappa' :> \kappa$.

4.1.2 Naive Interpretation

Given the above intent for the rollback primitive roll, how does one define its operational semantics? As hinted at in the introduction, this is actually a subtler affair than one may expect. A big difference with $\rho\pi$, where communication steps are undone one by one, is that the k in $\mathsf{roll}\ k$ may refer to a communication step far in the past. So the idea behind a process $\mathsf{roll}\ k$, referring to a memory $[M;k]$, is to restore the configuration that generated the memory and delete all its effects on the global configuration. Said otherwise, to restore the content of the memory we need to annul all its forward history, that is all the communications that have been generated *because of* it. Let us suppose having the following predicates. $N \blacktriangleright k$ states that all the active threads and memories in N bear tags κ that have k as causal antecedent, i.e., $k :> \kappa$. The predicate $\texttt{complete}(M_c)$ states that configuration M_c gathers all the threads (inside or outside memories) whose tags have as a causal antecedent the tag of a memory in M_c itself, i.e., if a memory in M_c is of the form $[M';k']$ (the communication M' created a process tagged with k'), then a process or a memory containing a process tagged with k' has to be in M_c (M_c contains every related process). Consider now the following attempt at a rule for roll:

$$\textsc{(Naive)}\quad \frac{N \blacktriangleright k \qquad \texttt{complete}(N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k))}{N \mid [M;k] \mid (\kappa : \mathsf{roll}\ k) \rightsquigarrow M \mid N\lightning_k}$$

The premises of rule $\textsc{Naive}$ thus asserts that the configuration $M_c = N \mid [M;k] \mid \kappa : \mathsf{roll}\ k$, on the left hand side of the reduction in the conclusion of the rule, gathers all (and only) the threads and memories which have originated from the process tagged by k, itself created by the interaction of the message and trigger recorded in M. Being complete, M_c is thus ready to be rolled-back and replaced by the configuration M which is at its origin, since there exist no other processes outside M_c caused by k. Rolling-back M_c has another effect, noted as $N\lightning_k$ in the right hand side of the conclusion, which is to release from memories those messages or triggers which do not have k as a causal antecedent, but which participated in communications with causal descendants of k.

For instance, the configuration $M_0 = M_1 \mid (\kappa_2 : c(Y) \triangleright_\delta Y)$, where $M_1 = (\kappa_0 : a\langle P\rangle) \mid (\kappa_1 : a(X) \triangleright_\gamma c\langle \mathsf{roll}\ \gamma\rangle)$, has the following forward

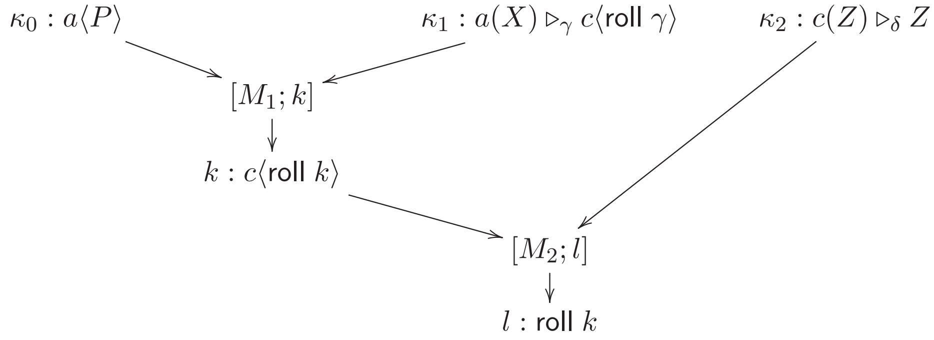
$\kappa_0 : a\langle P\rangle$
$\kappa_1 : a(X) \rhd_\gamma c\langle \mathsf{roll}\ \gamma\rangle$
$\kappa_2 : c(Z) \rhd_\delta Z$
$[M_1; k]$
$k : c\langle \mathsf{roll}\ k\rangle$
$[M_2; l]$
$l : \mathsf{roll}\ k$

Figure 4.1: Example of causal dependence graph.

reductions (where $M_2 = (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \rhd_\delta Y)$):

$$\begin{aligned} M_0 &\twoheadrightarrow \nu k.\, [M_1; k] \mid (k : c\langle \mathsf{roll}\ k\rangle) \mid (\kappa_2 : c(Y) \rhd_\delta Y) \\ &\twoheadrightarrow \nu k, l.\, [M_1; k] \mid [M_2; l] \mid (l : \mathsf{roll}\ k) = M_3 \end{aligned}$$

The causal dependence graph on tags of M_3 is depicted in Figure 4.1, where arrows may be read as *has caused*. As one can see, to undo the communication that caused k, that is M_1, we have to undo the communication M_2 and its effect tagged by l. Applying rule NAIVE (and structural congruence, defined later) on M_3 we get:

$$M_3 \rightsquigarrow M_1 \mid [M_2; l]\lightning_k = M_1 \mid (\kappa_2 : c(Y) \rhd_\delta Y) = M_0$$

where the process $(\kappa_2 : c(Y) \rhd_\delta Y)$ is released from memory $[M_2; l]$ because it does not depend on k, as show by Figure 4.1.

4.1.3 Concurrent Rolls

Rule NAIVE looks reasonable enough, but difficulties arise when concurrent rollbacks are taken into account. Consider the following configuration:

$$M = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0}\rangle)$$

where[1] $\tau_1 = a(X) \rhd_\gamma d\langle \mathbf{0}\rangle \mid (c(Y) \rhd \mathsf{roll}\ \gamma)$ and $\tau_3 = b(Z) \rhd_\delta c\langle \mathbf{0}\rangle \mid (d(U) \rhd \mathsf{roll}\ \delta)$.

Reductions of M are depicted in Figure 4.2. Forward reductions are labelled by the name of the channel used for communication, while backward reductions are labelled by the executed roll instruction. Processes and short-

[1] We assume parallel composition has precedence over trigger.

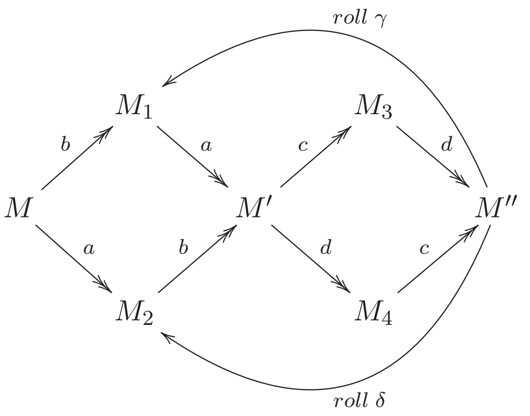

Figure 4.2: Concurrent rollback anomaly.

cuts of Figure 4.2 are detailed below:

$$\begin{aligned}
M_1 &= \nu l_2, h_3, h_4.\, \sigma_1 \mid [\sigma_2; l_2] \mid (\kappa_3 : c\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4)\\
M_2 &= \nu l_1, h_1, h_2.\, [\sigma_1; l_1] \mid (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_2 : \tau_2) \mid \sigma_2\\
M' &= \nu l_1, l_2, \tilde{h}.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid (\kappa_1 : d\langle \mathbf{0} \rangle) \mid \sigma_3 \mid \sigma_4\\
M_3 &= \nu l_1, l_2, l_3, \tilde{h}.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid (l_3 : \mathsf{roll}\ l_1) \mid \sigma_4\\
M_4 &= \nu l_1, l_2, l_4, \tilde{h}.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_4; l_4] \mid (l_4 : \mathsf{roll}\ l_2) \mid \sigma_3\\
M'' &= \nu \tilde{l}, \tilde{h}.\, [\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2)
\end{aligned}$$

with:

$$\begin{array}{lll}
\sigma_1 = (k_2 : a\langle \mathbf{0} \rangle) \mid (k_1 : \tau_1) & \sigma_2 = (k_4 : b\langle \mathbf{0} \rangle) \mid (k_3 : \tau_3) & \tau_2 = c(Y) \triangleright \mathsf{roll}\ l_1\\
\sigma_3 = (\kappa_3 : c\langle \mathbf{0} \rangle) \mid (\kappa_2 : \tau_2) & \sigma_4 = (\kappa_1 : d\langle \mathbf{0} \rangle) \mid (\kappa_4 : \tau_4) & \tau_4 = d(U) \triangleright \mathsf{roll}\ l_2\\
\kappa_1 = \langle h_1, \{h_1, h_2\} \rangle \cdot l_1 & \kappa_2 = \langle h_2, \{h_1, h_2\} \rangle \cdot l_1 & \tilde{l} = \{l_1, \ldots, l_4\}\\
\kappa_3 = \langle h_3, \{h_3, h_4\} \rangle \cdot l_2 & \kappa_4 = \langle h_4, \{h_3, h_4\} \rangle \cdot l_2 & \tilde{h} = \{h_1, \ldots, h_4\}
\end{array}$$

In M'' there are two enabled roll instructions aiming to two different and unrelated memories. Ideally one may think that from M'' it is possible to get back to M. The anomaly here is that the execution of one roll disables the other one, as one can see in the Figure 4.2 from M'' we can get back either to M_1 or to M_2, but there is no way, from M_1 or M_2 to get back to the initial configuration M. On the other hand, if we were in $\rho\pi$ from M_1 and M_2 it is possible to get back to M. Thus rule NAIVE is not unsound, but incomplete or insufficiently permissive, at least with respect to what is possible in $\rho\pi$. If we were to undo actions in M'' step by step, using $\rho\pi$'s backward reductions, we could definitely reach all of M, M_1, and M_2. Note that this issue is not due to the higher-order aspect of the calculus, we would have the same problem also in a reversible first order π-calculus.

The main motivation to have a complete rule comes from the fact that, in an abstract semantics, one wants to be as liberal as possible, and not unduly restrict implementations. If we were to pick the NAIVE rule as our semantics for rollback, then a correct implementation would have to enforce the same restrictions with respect to states reachable from backward reductions, restrictions which, in the case of rule NAIVE, are both complex to characterize (in terms of conflicting rollbacks) and quite artificial since they do not correspond to any clear execution policy. In the next section, we present a *maximally permissive* semantics for rollback, using $\rho\pi$ as our benchmark for completeness.

4.2 Syntax and Semantics

Names, keys, and variables. We assume the existence of the following denumerable infinite mutually disjoint sets: the set $\mathcal{N}$ of *names*, the set $\mathcal{K}$ of *keys*, the set $\mathcal{V}_{\mathcal{K}}$ of *tag variables*, and the set $\mathcal{V}_{\mathcal{P}}$ of process variables. The set $\mathcal{I} = \mathcal{N} \cup \mathcal{K}$ is called the set of *identifiers*. We denote by $\mathbb{N}$ the set of natural integers. We let (together with their decorated variants): a, b, c range over $\mathcal{N}$; h, k, l range over $\mathcal{K}$; u, v, w range over $\mathcal{I}$; δ, γ range over $\mathcal{V}_{\mathcal{K}}$; X, Y, Z range over $\mathcal{V}_{\mathcal{P}}$. We denote by $\tilde{u}$ a finite set of identifiers $\{u_1, \ldots, u_n\}$.

Syntax. The syntax of the roll-π calculus is given in Figure 4.3 (we often add balanced parenthesis around roll-π terms to disambiguate them). *Processes*, given by the P, Q productions in Figure 4.3, are the standard processes of the asynchronous Higher-Order π-calculus, except for the presence of the roll primitive and the extra bound tag variable in triggers. A trigger in roll-π takes the form $a(X) \triangleright_\gamma P$, which allows the receipt of a message of the form $a\langle Q\rangle$ on channel a, and the capture of the tag of the receipt event with tag variable γ.

Processes in roll-π cannot directly execute, only *configurations* can. *Configurations* in roll-π are given by the M, N productions in Figure 4.3. A configuration is built up from *tagged processes* and *memories*.

In a tagged process $\kappa : P$ the tag κ is either a single key k or a pair of the form $\langle h, \tilde{h}\rangle \cdot k$, where $\tilde{h}$ is a set of keys with $h \in \tilde{h}$. A tag serves as an identifier for a process. A *marked memory* is a configuration of the form $[\mu; k]^\bullet$, which just serves to indicate that a rollback operation targeting this memory has been initiated.

$$
\begin{array}{rlr}
P, Q ::= & & \textit{processes} \\
& \mathbf{0} & \textit{null process} \\
& \mid\ X & \textit{process variable} \\
& \mid\ \nu a.\, P & \textit{restriction} \\
& \mid\ (P \mid Q) & \textit{parallel} \\
& \mid\ a\langle P\rangle & \textit{message} \\
& \mid\ a(X) \triangleright_\gamma P & \textit{trigger} \\
& \mid\ \mathsf{roll}\ k & \textit{active roll} \\
& \mid\ \mathsf{roll}\ \gamma & \textit{roll}
\end{array}
$$

$$
\begin{array}{rlr}
M, N ::= & & \textit{configurations} \\
& \mathbf{0} & \textit{null configuration} \\
& \mid\ \nu u.\, M & \textit{restriction} \\
& \mid\ (M \mid N) & \textit{parallel} \\
& \mid\ \kappa : P & \textit{thread} \\
& \mid\ [\mu; k] & \textit{memory} \\
& \mid\ [\mu; k]^\bullet & \textit{marked memory}
\end{array}
$$

$$
\begin{array}{rlr}
\kappa ::= & k \ \mid\ \langle h, \tilde{h}\rangle \cdot k & \textit{tags} \\
\mu ::= & ((\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright Q)) & \textit{configuration part}
\end{array}
$$

$$
\begin{array}{r}
u \in \mathcal{I} \\
a \in \mathcal{N} \\
X \in \mathcal{V}_\mathcal{P} \\
h, k \in \mathcal{K} \\
\kappa \in \mathcal{T} \\
\gamma \in \mathcal{V}_\mathcal{K}
\end{array}
$$

Figure 4.3: Syntax of roll-π.

We denote by $\mathcal{P}$ the set of $\mathsf{roll}\text{-}\pi$ processes, and $\mathcal{C}$ the set of $\mathsf{roll}\text{-}\pi$ configurations. We call *agent* an element of the set $\mathcal{A} = \mathcal{P} \cup \mathcal{C}$. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$; L, M, N range over $\mathcal{C}$; and A, B, C range over $\mathcal{A}$. We call *thread*, a process that is either a message $a\langle P\rangle$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction $\mathsf{roll}\ k$. We let τ and its decorated variants range over threads. An immediate difference with respect to $\rho\pi$ is that now threads comprise also roll instructions.

Free identifiers and free variables. Notions of free identifiers and free variables in $\mathsf{roll}\text{-}\pi$ are usual. Constructs with binders are of the following forms: $\nu a.\, P$ binds the name a with scope P; $\nu u.\, M$ binds the identifier u with scope M; and $a(X) \triangleright_\gamma P$ binds the process variable X and the tag variable γ with scope P. We note $\mathtt{fn}(P)$, $\mathtt{fn}(M)$, and $\mathtt{fn}(\kappa)$ the set of free names, free identifiers, and free keys, respectively, of process P, of configuration M, and of tag κ. Note in particular that $\mathtt{fn}(\kappa : P) = \mathtt{fn}(\kappa) \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$, $\mathtt{fn}(k) = \{k\}$ and $\mathtt{fn}(\langle h, \tilde{h}\rangle \cdot k) = \tilde{h} \cup \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or tag) variable. We note $\mathcal{P}^{cl}$, $\mathcal{C}^{cl}$ and $\mathcal{A}^{cl}$ the sets of closed processes, configurations, and agents, respectively.

Initial and consistent configurations. Not all configurations allowed by the syntax in Figure 4.3 are meaningful. For instance, in a memory $[\mu; k]$, tags occurring in the configuration part μ must be different from the key k; if a tagged process $\kappa_1 : \mathsf{roll}\ k$ occurs in a configuration M, we expect a memory $[\mu; k]$ to occur in M as well. In the rest of the chapter, we only will be considering well-formed, or *consistent*, closed configurations. A configuration is consistent if it can be derived using the rules of the calculus from an *initial* configuration. A configuration is initial if it does not contain memories, all the tags are distinct and simple (i.e., of the form k), and the argument of each roll is bound by a trigger.

The definition of consistent configuration is similar to the Definition 3.1 where we take into account also roll processes as threads, and we modify the condition (6). We defer the formal definition of consistent configuration to the next section.

Remark 4.1 *We have no construct for replicated processes or guarded choice in $\mathsf{roll}\text{-}\pi$: as in HOπ, these can easily be encoded.*

Remark 4.2 *In the remainder of the chapter, we adopt* Barendregt's Variable Convention*: if terms $t_1, \ldots, t_n$ occur in a certain context (e.g., definition, proof),*

$$(\text{E.P}\textsc{ar}\text{C})\ A \mid B \equiv B \mid A \qquad (\text{E.P}\textsc{ar}\text{A})\ A \mid (B \mid C) \equiv (A \mid B) \mid C$$

$$(\text{E.P}\textsc{ar}\text{N})\ A \mid \mathbf{0} \equiv A \qquad (\text{E.N}\textsc{ew}\text{N})\ \nu u.\, \mathbf{0} \equiv \mathbf{0}$$

$$(\text{E.N}\textsc{ew}\text{C})\ \nu u.\, \nu v.\, A \equiv \nu v.\, \nu u.\, A \qquad (\text{E.N}\textsc{ew}\text{P})\ (\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$$

$$(\text{E.}\alpha)\ A =_\alpha B \implies A \equiv B \qquad (\text{E.T}\textsc{ag}\text{N})\ \kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$$

$$(\text{E.T}\textsc{ag}\text{P})\ k : \prod_{i=1}^{n} \tau_i \equiv \nu \tilde{h}.\, \prod_{i=1}^{n} (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Figure 4.4: Structural congruence for roll-π

then in these terms all bound identifiers and variables are chosen to be different from the free ones.

4.2.1 Operational semantics

The operational semantics of the roll-π calculus is defined via a reduction relation $\rightarrow$, which is a binary relation over closed configurations ($\rightarrow \subset \mathcal{C}^{cl} \times \mathcal{C}^{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over processes and configurations ($\equiv \subset \mathcal{P}^2 \cup \mathcal{C}^2$). We define *evaluation contexts* as "configurations with a hole $\cdot$", given by the following grammar:

$$\mathbb{E} ::= \cdot \ \mid \ (M \mid \mathbb{E}) \ \mid \ \nu u.\, \mathbb{E}$$

General contexts $\mathbb{C}$ are just processes or configurations with a hole $\cdot$. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that is closed for general contexts: $P \,\mathcal{R}\, Q \implies \mathbb{C}[P] \,\mathcal{R}\, \mathbb{C}[Q]$ or $M \,\mathcal{R}\, N \implies \mathbb{C}[M] \,\mathcal{R}\, \mathbb{C}[N]$.

The relation $\equiv$ is defined as the smallest congruence on processes and configurations that satisfies the rules in Figure 4.4. We note $t =_\alpha t'$ when terms t, t' are equal modulo α-conversion. If $\tilde{u} = \{u_1, \ldots, u_n\}$, then $\nu \tilde{u}.\, A$ stands for $\nu u_1.\, \ldots \nu u_n.\, A$. We note $\prod_{i=1}^{n} A_i$ for $A_1 \mid \ldots \mid A_n$ (there is no need to indicate how the latter expression is parenthesized because the parallel operator is associative by rule E.P<span style="font-variant:small-caps">ar</span>A). In rule E.TagP, processes τ_i are threads. Recall the use of the variable convention in these rules: for instance, in the rule $(\nu u.\, A) \mid B \equiv \nu u.\, (A \mid B)$ the variable convention makes implicit the condition $u \notin \texttt{fn}(B)$. The structural congruence rules are the usual rules for the π-calculus (E.ParC to E.α) without the rule dealing with replication, and with the addition of two new rules dealing with tags: E.TagN and E.TagP. Rule E.TagN is a scope extrusion rule to push restrictions to the top level. Rule E.TagP allows to generate unique tags for each thread in a configuration. An easy induction on the structure of terms provides us with a kind of normal form for configurations (by convention $\prod_{i \in I} A_i = \mathbf{0}$ if $I = \emptyset$, and $[\mu; k]^\circ$ stands for $[\mu; k]$ or $[\mu; k]^\bullet$):

Lemma 4.1 (Thread normal form) *For any configuration M, we have*

$$M \equiv \nu \tilde{u}. \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [\mu_j; k_j]^\circ$$

with $\rho_i = \mathbf{0}$, $\rho_i = \mathsf{roll}\ k_i$, $\rho_i = a_i \langle P_i \rangle$, or $\rho_i = a_i(X_i) \rhd_{\gamma_i} P_i$.

We can now formally define the notion of *consistent configuration.*

Definition 4.1 (Consistent Configuration) *A configuration $M \equiv \nu \tilde{u}. \prod_{i \in I} (\kappa_i : \rho_i) \mid \prod_{j \in J} [M_j; k_j]^\circ$, with $\rho_i = \mathbf{0}$ or ρ_i a primitive thread process, $M_j = \delta_j : R_j \mid \gamma_j : T_j$, $R_j = a_j \langle P_j \rangle$, $T_j = a_j(X_j) \rhd_{\gamma_i} Q_j$, is said to be* consistent *if the following conditions are met:*

- *it satisfies conditions* (1),(2),(3),(4),(5) *of Definition 3.1;*
- *condition* (6) *of Definition 3.1 is substituted with : For all $j \in J$, there exist $E \subseteq I$, $D \subseteq J \setminus \{j\}$, $G \subseteq J \setminus \{j\}$, such that:*

$$\nu \tilde{u}. k_j : Q_j \{^{P_j, k_j}/_{X_j, \gamma_j}\} \equiv \nu \tilde{u}. \prod_{e \in E} \kappa_e : \rho_e \mid \prod_{d \in D} \delta_d : R_d \mid \prod_{g \in G} \gamma_g : T_g$$

We say that a binary relation $\mathcal{R}$ on closed configurations is *evaluation-closed* if it satisfies the inference rules:

$$\text{(R.Ctx)}\ \frac{M \mathrel{\mathcal{R}} N}{\mathbb{E}[M] \mathrel{\mathcal{R}} \mathbb{E}[N]}$$

$$\text{(R.Eqv)}\ \frac{M \equiv M' \qquad M' \mathrel{\mathcal{R}} N' \qquad N' \equiv N}{M \mathrel{\mathcal{R}} N}$$

The reduction relation $\rightarrow$ is defined as the union of two relations, the *forward* reduction relation $\twoheadrightarrow$ and the *backward* reduction relation $\rightsquigarrow$: $\rightarrow = \twoheadrightarrow \cup \rightsquigarrow$. Relations $\twoheadrightarrow$ and $\rightsquigarrow$ are defined to be the smallest evaluation-closed binary relations on closed configurations satisfying the rules in Figure 4.5.

In the rest of the chapter when needed, we will refer to this semantics as HL (for High-Level) semantics.

$$\text{(H.Com)} \quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(H.Start)} \quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet$$

$$\text{(H.Roll)} \quad \frac{N \blacktriangleright k \qquad \mathtt{complete}(N \mid [\mu; k])}{N \mid [\mu; k]^\bullet \rightsquigarrow \mu \mid N\lightning_k}$$

Figure 4.5: Reduction rules for roll-π.

The rule for forward reduction H.Com is the standard communication rule of the Higher-Order π-calculus with three side effects: (i) the creation of a new memory to record the configuration that gave rise to it (note again the use of the variable convention: the key k is fresh); (ii) the tagging of the continuation of the message receipt with the fresh key k; (iii) the passing of the newly created tag k as a parameter to the newly launched instance of the trigger's body Q.

Backward reduction is subject to the rules H.Roll and H.Start. Rule H.Roll is similar to rule Naive defined in the previous section, except that it relies on the presence of a marked memory $[\mu; k]^\bullet$ instead of that of the process κ : roll k to roll-back a given configuration. Rule H.Start just marks a memory to enable rollback. By breaking down the atomicity of rule Naive into a two step execution (H.Start followed by H.Roll) we have a way around to solve the anomaly depicted in Figure 4.2, as will be shown later.

Causal Dependence

The way in which new keys are created in the rule H.Com provides us a notion of ordering on keys, that is we may state that in the forward rule keys κ_1 and κ_2 come *before* the new key k, or more precisely they cause the tag k. The definition of rule H.Roll exploits several predicates and relations which we define below.

Definition 4.2 (Causal dependence) *Let M be a configuration and let T_M be the set of tags occurring in M. The binary relation $>_M$ on T_M is defined as the smallest relation satisfying the following clauses:*

- $k >_M \langle h_i, \tilde{h}\rangle \cdot k$;

- *$\kappa' >_M k$ if κ' occurs in μ for some memory $[\mu; k]^\circ$ that occurs in M.*

The causal dependence relation $:>_M$ is the reflexive and transitive closure of $>_M$.

The relation $>$, on keys, states that a key k causes all its descendants generated by an application of the structural law E.TagP and that a key κ in the configuration part of a memory causes the memory tag. For example, let $\mu = (\kappa_1 : a\langle 0\rangle) \mid (\kappa_2 : a(X) \triangleright b\langle 0\rangle \mid c\langle 0\rangle)$, we have that $\mu \twoheadrightarrow M$, with:

$$M = \nu h_1, h_2, k.\, [\mu_1; k] \mid (\langle h_1, \{h_1, h_2\}\rangle \cdot k : a\langle 0\rangle) \mid (\langle h_2, \{h_1, h_2\}\rangle \cdot k : b\langle 0\rangle)$$

Then $k >_M \langle h_1, \{h_1, h_2\}\rangle \cdot k$ and $k >_M \langle h_2, \{h_1, h_2\}\rangle \cdot k$ and both κ_1 and κ_2 cause k, that is $\kappa_1 >_M k$ and $\kappa_2 >_M k$.

Relation $\kappa :>_M \kappa_1$ reads "κ is a causal antecedent of κ_1 according to M". When configuration M is clear from the context, we write $\kappa :> \kappa_1$ for $\kappa :>_M \kappa_1$. Let us note that the relation $:>$ is a partial *order relation* since **reflexivity** ($\kappa :> \kappa$), **antisymmetry** ($\kappa :> \kappa_1$ and $\kappa_1 :> \kappa$ implies $\kappa = \kappa_1$) and **transitivity** hold. Exploiting the $:>$ relation, we can easily state when a configuration M depends on a particular key κ, that is when all the tags contained in a configuration have as causal antecedent a particular key κ.

Definition 4.3 (κ dependence) *Let $M \equiv \nu\tilde{u}.\prod_{i\in I} \kappa_i : \rho_i \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$. Configuration M is κ-dependent, written $M \blacktriangleright \kappa$, if $\forall l \in I \cup J,\ \kappa :>_M \kappa_l$.*

We now define the *projection* operation on configurations $M \lightning_\kappa$, that captures the parallel composition of all tagged processes, occurring in memories in M, that do not depend on κ. This *filtering* operator is used to delete the global effects of a memory forward history, since it allows to put back in the environment processes not related with the ongoing rollback, that participated to communications with processes *caused* by the memory aimed by the ongoing rollback.

Definition 4.4 (Projection) *Let $M \equiv \nu\tilde{u}.\ \prod_{i\in I}(\kappa_i : \rho_i) \mid \prod_{j\in J}[\mu_j; \kappa_j]^\circ$, with $\mu_j = \kappa'_j : R_j \mid \kappa''_j : T_j$. Then:*

$$M \lightning_\kappa = \nu\tilde{u}.\, (\prod_{j'\in J'} \kappa'_{j'} : R_{j'}) \mid (\prod_{j''\in J''} \kappa''_{j''} : T_{j''})$$

where $J' = \{j \in J \mid \kappa \not:> \kappa'_j\}$ and $J'' = \{j \in J \mid \kappa \not:> \kappa''_j\}$.

Finally we define the notion of *complete* configuration, used in the premise of rule H.Roll.

Definition 4.5 (Complete configuration) *A configuration M* contains *a tagged process $\kappa : P$, written $\kappa : P \in M$, if $M \equiv \nu\tilde{u}.\,(\kappa : P) \mid N$ or $M \equiv \nu\tilde{u}.\,[\kappa : P \mid \kappa_1 : Q; k]^\circ \mid N$.*

A configuration M is complete*, noted* `complete`(M)*, if for each memory $[\mu; k]^\circ$ that occurs in M, one of the following holds:*

1. *There exists a process P such that $k : P \in M$.*

2. *There is $\tilde{h}$ such that for each $h_i \in \tilde{h}$ there exists a process P_i such that $\langle h_i, \tilde{h}\rangle \cdot k : P_i \in M$.*

A configuration M is `complete` if all the continuations of all the memories are contained in M itself. In this way, if $M \blacktriangleright k$ and `complete`$(M \mid [\mu; k])$ (premises of rule H.Roll) hold, we are sure that M contains all the *forward history* (or execution) of the memory $[\mu; k]$, and then we can restore μ by deleting its effects on M (conclusion of the of rule H.Roll).

We now consider again the example of concurrent interfering rolls used in Section 4.1 to show how the new semantics can cope with the problem depicted in Figure 4.2. Consider the following configuration:

$$M = (k_1 : \tau_1) \mid (k_2 : a\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) \mid (k_4 : b\langle \mathbf{0}\rangle)$$

where $\tau_1 = a(X) \triangleright_\gamma d\langle \mathbf{0}\rangle \mid (c(Y) \triangleright \mathsf{roll}\ \gamma)$ and $\tau_3 = b(Z) \triangleright_\delta c\langle \mathbf{0}\rangle \mid (d(U) \triangleright \mathsf{roll}\ \delta)$. Then a possible execution could be

$$\begin{array}{ll}
 & M \twoheadrightarrow \\
(1) & \nu\tilde{l}, \tilde{h}.\,[\sigma_1; l_1] \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2) \rightsquigarrow \\
(2) & \nu\tilde{l}, \tilde{h}.\,[\sigma_1; l_1]^\bullet \mid [\sigma_2; l_2] \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2) \rightsquigarrow \\
(3) & \nu\tilde{l}, \tilde{h}.\,[\sigma_1; l_1]^\bullet \mid [\sigma_2; l_2]^\bullet \mid [\sigma_3; l_3] \mid [\sigma_4; l_4] \mid (l_3 : \mathsf{roll}\ l_1) \mid (l_4 : \mathsf{roll}\ l_2) \rightsquigarrow \\
(4) & \nu l_2, h_1, h_2.\,\sigma_1 \mid [\sigma_2; l_2]^\bullet \mid (\kappa_3 : c\langle 0\rangle) \mid (\kappa_4 : \tau_4) \rightsquigarrow \\
(5) & \sigma_1 \mid \sigma_2 = M
\end{array}$$

with

$$\begin{array}{lll}
\sigma_1 = (k_2 : a\langle \mathbf{0}\rangle) \mid (k_1 : \tau_1) & \sigma_2 = (k_4 : b\langle \mathbf{0}\rangle) \mid (k_3 : \tau_3) & \tau_2 = c(Y) \triangleright \mathsf{roll}\ l_1 \\
\sigma_3 = (\kappa_3 : c\langle \mathbf{0}\rangle) \mid (\kappa_2 : \tau_2) & \sigma_4 = (\kappa_1 : d\langle \mathbf{0}\rangle) \mid (\kappa_4 : \tau_4) & \tau_4 = d(U) \triangleright \mathsf{roll}\ l_2 \\
\kappa_1 = \langle h_1, \{h_1, h_2\}\rangle \cdot l_1 & \kappa_2 = \langle h_2, \{h_1, h_2\}\rangle \cdot l_1 & \tilde{l} = \{l_1 \dots l_4\} \\
\kappa_3 = \langle h_3, \{h_3, h_4\}\rangle \cdot l_2 & \kappa_4 = \langle h_4, \{h_3, h_4\}\rangle \cdot l_2 & \tilde{h} = \{h_1 \dots h_4\}
\end{array}$$

As one can see, after that all the forward communications are done (1) by executing the two rule H.Start on the two enabled rolls, we can mark the two memories aimed by them. In this way, we can *remember* that there was a roll process targeting a particular memory. Once the two memories are marked (3), then we can apply twice the rule H.Roll and we can get back to the initial configuration M. So by using a marked memory as requirement for the rollback and not requiring the syntactical presence of the roll process that marked it, we can cope with the problem raised by the Naive rule. We break the atomicity of the Naive rule by allowing possible communications between the execution H.Start and the execution of H.Roll.

4.3 Soundness and completeness of backward reduction in roll-π

We present in this section a Loop Theorem, that establishes the soundness of backward reduction in roll-π, and we prove the completeness (or maximal permissiveness) of backward reduction in roll-π.

Theorem 4.1 (Loop Theorem - Soundness of backward reduction) *For any (consistent) configurations M and M' with no marked memories, if $M \rightsquigarrow^* M'$, then $M' \twoheadrightarrow^* M$.*

Proof: The computation $M \rightsquigarrow^* M'$ is composed by applications of rules H.Start and H.Roll. Also, if there is no H.Roll application then the computation is empty, since M' does not contain marked memories by hypothesis. We proceed by induction on the number of applications of H.Roll. The base case (zero applications) is trivial.

For the inductive case, take the last such application, which is also the last reduction in the computation, as M' has no marked memories: $M \rightsquigarrow^* M'' \rightsquigarrow M'$. Let us consider the computation obtained by removing from $M \rightsquigarrow^* M''$ the applications of rule H.Start that added marks removed by $M'' \rightsquigarrow M'$ (there is at least the mark corresponding to the rollback, with additionally other marks coming from causally dependent memories removed as a side effect). This is of the form $M \rightsquigarrow^* M''_1$ with M''_1 with no marked memories, and equal to M'' but for missing marks. By inductive hypothesis $M''_1 \twoheadrightarrow^* M$. We now need to prove that $M' \twoheadrightarrow^* M''_1$ to conclude. We prove the following property: for every M^i with at least one mark such that $M^i \rightsquigarrow M'$, and for every M^i_1 equal to M^i except for having no marks, we have $M' \twoheadrightarrow^* M^i_1$. We prove this property by induction on the number of memories removed by the step $M^i \rightsquigarrow M'$. If only one memory is removed, then we simply replay the communication it contained and have $M' \twoheadrightarrow M^i_1$. For the inductive case, we consider a removed memory whose only causal descendant is a process, i.e., M^i contains a sub-process of the form $[\mu; k']^\circ \mid (k' : P)$. Let M^- be the same configuration where this process has been replaced by μ (thus undoing the communication that created the memory). As there are at least two memories removed in $M^i \rightsquigarrow M'$, and as every removed memory is a causal descendant of M_k (the memory for which the rollback is done), the memory $[\mu; k']^\circ$ cannot be M_k (since M_k does not only have a process as causal descendant, but also at least one other memory). Thus we have $M^- \rightsquigarrow M'$, and by induction (one fewer memory is removed), we have $M' \twoheadrightarrow^* M_1^-$. As M_1^- is M^- with every mark removed, we have $M_1^- \twoheadrightarrow M_1^i$ by replaying the communication μ. Thus we have $M' \twoheadrightarrow^* M_1^i$. We conclude by applying this property to $M'' \rightsquigarrow M'$, thus we have $M' \twoheadrightarrow^* M_1''$. The thesis follows. □

To state the completeness result for backward reduction in roll-π, we define a family of functions $\phi_e : \mathcal{C}_{\mathsf{roll}\text{-}\pi} \to \mathcal{C}_{\rho\pi}$, where $e \in \mathcal{N}$, mapping a roll-π configuration to a $\rho\pi$ configuration. Function ϕ_e is defined by induction as follows:

$$\begin{array}{rcl}
\phi_e(\nu u.\, M) = \nu u.\, \phi_e(M) & \qquad & \phi_e(M \mid N) = \phi_e(M) \mid \phi_e(N) \\
\phi_e(\kappa : P) = \kappa : \phi_e(P) & & \phi_e([\mu; k]^\circ) = [\phi_e(\mu); k] \\
\phi_e(\nu a.\, P) = \nu a.\, \phi_e(P) & & \phi_e(P \mid Q) = \phi_e(P) \mid \phi_e(Q) \\
\phi_e(\mathsf{roll}\ k) = e\langle \mathbf{0} \rangle & & \phi_e(\mathsf{roll}\ \gamma) = e\langle \mathbf{0} \rangle \\
\phi_e(a\langle P \rangle) = a\langle \phi_e(P) \rangle & & \phi_e(a(X) \triangleright_\gamma P) = a(X) \triangleright \phi_e(P) \\
\phi_e(\mathbf{0}) = \mathbf{0} & & \phi_e(X) = X
\end{array}$$

Note that roll instructions are transformed not into $\mathbf{0}$ but into a thread $e\langle \mathbf{0} \rangle$: this is to ensure a consistent roll-π configuration is transformed into a consistent $\rho\pi$ configuration (recall that $\mathbf{0}$ is not a thread, thus it may be collected by structural congruence and there would be no thread corresponding to the roll k process).

We now state that roll-π is maximally permissive: any subset of roll primitives in evaluation context may successfully be executed, unlike the naive example of Section 4.1.2. Let $M = \nu \tilde{u}.\, [\mu; k] \mid (k : P) \mid N$ be a $\rho\pi$ configuration and $S = \{k_1, \ldots, k_n\}$ a set of keys. We note $M \rightsquigarrow_S M'$ if $M \rightsquigarrow_{\rho\pi} M'$, $M' = \nu \tilde{u}.\, \mu \mid N$, and $k_i :> k$ for some $k_i \in S$ (here k is the key of the memory $[\mu; k]$ consumed by the backward reduction). If $M' \not\rightsquigarrow_S$, we say that M' is *final with respect to S*. We note $\rightsquigarrow_S^*$ the reflexive and transitive closure of $\rightsquigarrow_S$. Essentially $\rightsquigarrow_S$ restricts $\rho\pi$ backward reductions to processes that have been caused by at least one of the keys present in the set S. We assume here that reductions are name-preserving, i.e., existing keys are not

α-converted (see Definition 3.5 in Section 3.5).

Definition 4.6 (Marks removing) *Be $\zeta : \mathcal{C}_{\textsf{roll-}\pi} \to \mathcal{C}_{\textsf{roll-}\pi}$ defined by induction as follows:*

$$\begin{aligned} \zeta(\nu u.\, M) &= \nu u.\, \zeta(M) & \zeta(M \mid N) &= \zeta(M) \mid \zeta(N) \\ \zeta(\kappa : P) &= \kappa : P & \zeta([\mu; k]^{\circ}) &= [\mu; k] \\ \zeta(\mathbf{0}) &= \mathbf{0} \end{aligned}$$

Function ζ is just used to delete marks from a configuration.

Theorem 4.2 (Completeness of backward reduction) *Let M be a (consistent) $\textsf{roll-}\pi$ configuration such that $M \equiv \nu \tilde{u}.\prod_{i=1}^{n} \kappa_i : \textsf{roll}\ k_i \mid M_1$, let $S = \{k_1, \ldots, k_n\}$, and let $e \in \mathcal{N} \backslash \mathtt{fn}(M)$. Then for all $T \subseteq S$, if $\phi_e(M) \rightsquigarrow_T^* N$ and N is final with respect to T, there exists M' such that $N = \phi_e(M')$, and $M \rightsquigarrow_{\textsf{roll-}\pi}^* M'$.*

Proof: Let us consider the computation obtained in $\textsf{roll-}\pi$ by starting from M and first applying rule H.Start for every unmarked memory with key in T, yielding configuration M_m, then applying rule H.Roll for every marked memory with key in T (in an arbitrary order). Such a computation has the form $M \rightsquigarrow_{\textsf{roll-}\pi}^* M'$. Hence we have the following $\textsf{roll-}\pi$ execution:

$$M \rightsquigarrow_{\text{H.Start}} M_1 \rightsquigarrow_{\text{H.Start}} \cdots \rightsquigarrow_{\text{H.Start}} M_m \rightsquigarrow_{\text{H.Roll}} \cdots \rightsquigarrow_{\text{H.Roll}} M'$$

in this way all the memories whose key is in T are marked in M_m, and rolled-back in M'. We now show that this M' is the candidate one.

Let $M_u = \zeta(M)$, we have $M_u \rightsquigarrow_{\textsf{roll-}\pi}^* M'_u$, where $M'_u = \zeta(M')$, by the following reasoning: first every memory with key k_i in T has a $\textsf{roll}\ k_i$ in M_u to mark the memory using rule H.Start. Then we apply the same sequence of reductions as from M_m to M' (the fact that some memories which are not rolled-back are marked does not prevent any reduction). The resulting configuration has no marks (as every marked memory has been rolled-back), and only lacks some marks when compared to M'.

By the Loop Theorem (Theorem 4.1), we have $M'_u \twoheadrightarrow_{\textsf{roll-}\pi}^* M_u$. Since $\phi_e(M_u) = \phi_e(M)$ and $\phi_e(M'_u) = \phi_e(M')$, the forward computation $M'_u \twoheadrightarrow_{\textsf{roll-}\pi}^* M_u$ can be translated into $\rho\pi$: $\phi_e(M') \twoheadrightarrow_{\rho\pi}^* \phi_e(M)$. From the Loop Lemma of $\rho\pi$ (Lemma 3.11 in Section 3.3), we also have $\phi_e(M) \rightsquigarrow_{\rho\pi}^* \phi_e(M')$. Note that all the reductions involve memories which are descendants of keys in T and that $\phi_e(M')$ is final with respect to T.

Let us take an arbitrary $\rho\pi$ computation $\phi_e(M) \rightsquigarrow_T^* N$ such that N is final with respect to T. We will show that $\phi_e(M') \equiv N$. This will prove the thesis.

The proof is by induction on the number of reductions in $\phi_e(M) \rightsquigarrow_T^* N$. The base case (zero reductions) is trivial, as it means that $\phi_e(M)$ was already final with respect to T, thus (by consistency of M), $T = \emptyset$ and $M' = M$. For the inductive case consider the first reduction in both $\phi_e(M) \rightsquigarrow_T^* N$ and $\phi_e(M) \rightsquigarrow_{\rho\pi}^* \phi_e(M')$ (they both have one since they are not final with respect to T). If the two reductions involve the same memory then they coincide, and the thesis follows by inductive hypothesis. Otherwise, let $[\mu, k]$ be the memory involved in the first reduction for $\phi_e(M) \rightsquigarrow_T^* N$. The same memory is involved in a reduction in $\phi_e(M) \rightsquigarrow_{\rho\pi}^* \phi_e(M')$ (otherwise $\phi_e(M')$ would not be final). Note now that two enabled backward reductions can always be swapped (by Lemma 3.13 in Section 3.5). By applying multiple times this swapping, one can move the reduction involving $[\mu, k]$ to the beginning of $\phi_e(M) \rightsquigarrow_{\rho\pi}^* \phi_e(M')$ without changing the final state. Thus we are back to the case where the initial reduction is the same and we can apply the inductive hypothesis. □

4.4 Intermediate Semantics

We have shown that roll-π is sound (Theorem 4.1) and complete (Theorem 4.2) with respect to backward reductions. This implies that its semantics is the one we had in mind for controlling rollback. But as one can see from the semantics rules in Figure 4.5 the rule H.Roll implies several global checks (κ-dependency and completeness) on the environment. Moreover, the operation of restoring the content of a memory by erasing its effect on the environment is a big atomic step involving an unbounded number of processes. Hence this semantics, even if sound and complete, is far away from an actual implementation. Therefore, in the rest of the chapter we propose several refinements of the roll-π semantics toward a distributed low level algorithm, relying just on asynchronous runtime notifications and local checks.

A rollback operation (of a marked memory) can be divided into two phases: the first one consists of collecting all processes caused by the memory and that will be affected by the rollback, and the second phase consisting of restoring the content of the memory and deleting all the related processes. Moreover, we can see a memory and all the processes that it has caused as a tree rooted in the memory itself. So the collecting phase is similar to a *top down* visit on a tree where processes are leaves and memories are nodes. While visiting the tree, processes related to the aimed memory are *marked*. This idea is behind the low level semantics that will be presented in Section 4.5. To prove the correspondence between the low level semantics and the high level one (Theorem 4.7) our strategy is to introduce different semantics, each one being a refinement of the previous one, differing just by slight changes. Proving that two *almost similar* semantics are equivalent and then concatenate all these results by transitivity, is easier (even if laborious) than trying to find a relation (bisimulation) that directly relates the high level semantics with the low level one. We now list the different semantics, explaining how the two phases rollback is performed in each of them.

Freezing Semantics (FR): the first phase is done as a top down visit, but in a synchronous way: when a collected process or memory is in parallel with a direct descendant of it, the descendant can be collected. Once all the required processes have been collected, the deleting phase is still atomic.

Roll Semantics (RL): the top down visit is performed by using *runtime asynchronous notifications*. Once all the required processes have been collected, the deleting phase is still atomic.

Distributed Semantics (DS): the top down visit is done by asynchronous notifications (as in the previous semantics). The deleting phase is no longer atomic: it is done in several steps as in $\rho\pi$, each one undoing a single communication.

Low Level Semantics (LL): similar to the the DS one but using simpler runtime roll notifications. This semantics will be presented in Section 4.5.

The reader that is not willing to get all the technical details of the proof of Theorem 4.7, can easily skip this section and continue directly with Section 4.5, where the low level (LL) semantics is introduced.

Before introducing the various refinements we define a kind of behavioural equivalence able to relate two different semantics.

4.4.1 Contextual equivalence in roll-π

The operational semantics of the roll-π calculus is completed classically by the definition of a contextual equivalence between configurations, which takes the form of a barbed congruence. We first define observables in configurations. We say that name a is *observable in configuration* M, noted $M \downarrow_a$, if $M \equiv \nu\tilde{u}.\,(\kappa : a\langle P\rangle) \mid N$, with $a \notin \tilde{u}$. Keys are not observable, because they are just an internal device used to support reversibility. We note $\Rightarrow$, $\twoheadrightarrow^*$, $\rightsquigarrow^*$ the reflexive and transitive closures of $\rightarrow$, $\twoheadrightarrow$, and $\rightsquigarrow$, respectively.

One of the aims of this chapter is to define a low-level semantics for roll-π, close to a real distributed algorithm, and show that it is equivalent to the high-level one. We use weak barbed congruence for this purpose. Thus we need a definition of barbed congruence able to relate roll-π configurations executed under different semantics. These semantics will also rely on different runtime syntaxes. Thus, we define a family of relations, each labelled by the semantics to be used on the left and right components of its elements. We also label sets of configurations with the corresponding semantics, thus highlighting that the corresponding runtime syntax has to be included. However, contexts do not include runtime syntax, since we never add contexts at runtime.

Definition 4.7 (Barbed bisimulation and congruence) *A relation* $_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ *on closed consistent configurations is a* strong *(resp.* weak*)* barbed simulation *if whenever* $M\ _{s1}\mathcal{R}_{s2}\ N$

- $M \downarrow_a$ *implies* $N \downarrow_a$ *(resp.* $N \Rightarrow_{s2}\downarrow_a$*)*
- $M \rightarrow_{s1} M'$ *implies* $N \rightarrow_{s2} N'$*, with* $M'\,_{s1}\mathcal{R}_{s2}N'$ *(resp.* $N \Rightarrow_{s2} N'$ *with* $M'\,_{s1}\mathcal{R}_{s2}N'$*)*

A relation $_{s1}\mathcal{R}_{s2} \subseteq \mathcal{C}^{cl}_{s1} \times \mathcal{C}^{cl}_{s2}$ *is a* strong *(resp.* weak*)* barbed bisimulation *if* $_{s1}\mathcal{R}_{s2}$ *and* $(_{s1}\mathcal{R}_{s2})^{-1}$ *are strong (resp. weak) barbed simulations. We call* strong *(resp.* weak*)* barbed bisimilarity *and note* $_{s1}{\sim}_{s2}$ *(resp.* $_{s1}{\approx}_{s2}$*) the largest strong (resp. weak) barbed bisimulation with respect to semantics* $s1$ *and* $s2$*.*

We say that two configurations M *and* N *are* strong *(resp.* weak*)* barbed congruent*, written* $_{s1}\overset{\cdot\ c}{\sim}_{s2}$ *(resp.* $_{s1}\overset{\cdot\ c}{\approx}_{s2}$*), if for each* roll-$\pi$ *context* $\mathbb{C}$ *such that* $\mathbb{C}[M]$ *and* $\mathbb{C}[N]$ *are consistent, then* $\mathbb{C}[M]\,_{s1}{\sim}_{s2}\,\mathbb{C}[N]$ *(resp.* $\mathbb{C}[M]\,_{s1}{\approx}_{s2}\,\mathbb{C}[N]$*).*

Lemma 4.2 *If* $M_1\,_{s1}{\approx}_{s2}\,M_2$ *and* $M_2\,_{s2}{\approx}_{s3}\,M_3$ *then* $M_1\,_{s1}{\approx}_{s3}\,M_3$*.*

Proof: Let $\mathcal{R} = \{(M_1, M_3) \mid \exists\, M_2\,.M_1\,_{s1}{\approx}_{s2}\,M_2\,_{s2}{\approx}_{s3}\,M_3\}$. We show that $\mathcal{R} \subseteq\ _{s1}{\approx}_{s3}$.

Let us check barbs. By definition of $_{s1}{\approx}_{s2}$ if $M_1 \downarrow_a$ then $M_2 \Rightarrow_{s2} M_2' \downarrow_a$. Since $M_2\,_{s2}{\approx}_{s3}\,M_3$ this implies that if $M_2 \Rightarrow_{s2} M_2'$ then $M_3 \Rightarrow_{s3} M_3'$ and since $M_2' \downarrow_a$ then $M_3' \Rightarrow M_3'' \downarrow_a$. Hence $M_1 \downarrow_a$ implies that $M_3 \Rightarrow M_3'' \downarrow_a$. We can apply the same reasoning to the opposite case.

Let us consider reduction challenges. If $M_1 \rightarrow_{s1} M_1'$ then by definition $M_2 \Rightarrow_{s2} M_2'$ with $M_1' \,{}_{s1}{\approx}_{s2}\, M_2'$, but $M_2 \Rightarrow_{s2} M_2'$ implies that $M_3 \Rightarrow_{s3} M_3'$ such that $M_2' \,{}_{s2}{\approx}_{s3}\, M_3'$. Hence we have that $M_1 \rightarrow_{s1} M_1'$ implies that $M_3 \Rightarrow_{s3} M_3'$ with $(M_1', M_3') \in \mathcal{R}$ as desired. We can apply the same reasoning to the opposite case. □

Lemma 4.3 *If $M_1 \,{}_{s1}\dot{\approx}^{c}_{s2}\, M_2$ and $M_2 \,{}_{s2}\dot{\approx}^{c}_{s3}\, M_3$ then $M_1 \,{}_{s1}\dot{\approx}^{c}_{s3}\, M_3$.*

Proof: By definition we have that $\forall\, \mathbb{C}.\ \mathbb{C}[M_1] \,{}_{s1}{\approx}_{s2}\, \mathbb{C}[M_2] \,{}_{s2}{\approx}_{s3}\, \mathbb{C}[M_3]$, with $\mathbb{C}$ a roll-π context. By applying Lemma 4.2 we obtain that $\forall\, \mathbb{C}.\ \mathbb{C}[M_1] \,{}_{s1}{\approx}_{s2}\, \mathbb{C}[M_2] \,{}_{s2}{\approx}_{s3}\, \mathbb{C}[M_3]$ implies that $\forall\, \mathbb{C}.\ \mathbb{C}[M_1] \,{}_{s1}{\approx}_{s3}\, \mathbb{C}[M_3]$, hence $M_1 \,{}_{s1}\dot{\approx}^{c}_{s3}\, M_3$, as desired. □

4.4.2 Freezing Semantics

The atomicity of rule H.Roll comes from the fact that at once the rule gathers all the forward history of a particular memory $[\mu; k]$ and then restores it by deleting its effects. Our first refinement consists in breaking down the atomicity of this gathering action. We introduce, in the semantics, a way of collecting k-dependant processes similar to a tree visit: starting from the target memory of a roll (representing the root), step by step we visit the causal tree and in the meantime we mark the visited process as *being part* of a roll operation. Figure 4.6 shows the syntax of the FR (for freezing) refinement. The syntax is similar to that of roll-π (Figure 4.3) with the slight difference that now a configuration M can be a frozen process of the form $\lfloor \kappa : P \rfloor_S$. Moreover, μ configurations now can contain frozen processes. A process of the form $\lfloor \kappa : P \rfloor_S$ (with $S \neq \emptyset$) is a frozen process that is participating to at least one roll operation. The set S of keys, keeps track of all the rolls interested in that particular process. Frozen processes are blocked: they cannot perform communications.

$$\begin{array}{rcl} P, Q & ::= & \ldots \\ M, N & ::= & \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid \kappa : P \mid \lfloor \kappa : P \rfloor_S \mid [\mu; k] \mid [\mu; k]^{\bullet} \end{array}$$

Figure 4.6: Syntax of FR refinement.

Before giving the semantics of the FR refinement, we need to define a few operators.

Definition 4.8 (k-labelled) *A memory M is k-labelled if it is of the form $[\lfloor \kappa_1 : P \rfloor_S \mid M'; k_2]$ with $k \in S$. A process $\lfloor \kappa : P \rfloor_S$ is k-labelled if $k \in S$.*

Definition 4.9 ((k, k_1)-labelling) *A memory M is (k, k_1)-labelling if:*

1. *$k = k_1$ and M has the form $[\mu'; k]^\bullet$ or;*
2. *M has the form $[\mu; k_1]^\circ$ and it is k-labelled*

$$\text{(F.Com)}\quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow \nu k.\,(k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(F.Start)}\quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{FR} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^\bullet$$

$$\text{(F.InP)}\quad \frac{M \text{ is } (k, k_1) - \text{labeling} \quad \kappa_1 = k_1 \vee \kappa_1 = \langle h_i, \tilde{h}\rangle \cdot k_1 \quad k \notin S}{M \mid \lfloor \kappa_1 : P \rfloor_S \rightsquigarrow_{FR} M \mid \lfloor \kappa_1 : P \rfloor_{S \cup \{k\}}}$$

$$\text{(F.InM)}\quad \frac{M \text{ is } (k, k_1) - \text{labeling} \quad \kappa_1 = k_1 \vee \kappa_1 = \langle h_i, \tilde{h}\rangle \cdot k_1 \quad k \notin S}{M \mid [\lfloor \kappa_1 : P \rfloor_S \mid M; k_2]^\circ \rightsquigarrow_{FR} M \mid [\lfloor \kappa_1 : P \rfloor_{S \cup \{k\}} \mid M; k_2]^\circ}$$

$$\text{(F.Roll)}\quad \frac{\mathtt{complete}([\mu; k] \mid \prod_{i \in I} N_i) \quad \forall i \in I. N_i \text{ is } k\text{-labelled}}{[\mu; k]^\bullet \mid \prod_{i \in I} N_i \rightsquigarrow_{FR} \mu \mid (\prod_{i \in I} N_i)\lightning_k}$$

$$\text{(E.Freeze)}\quad \kappa : P \equiv_{FR} \lfloor \kappa : P \rfloor_\emptyset$$

Figure 4.7: FR semantics.

The FR semantics $\rightarrow_{FR}$ of roll-π is defined as for the HL one (cf. Section 4.2.1), as $\rightarrow_{FR} = \twoheadrightarrow_{FR} \cup \rightsquigarrow_{FR}$, where relations $\twoheadrightarrow_{FR}$ and $\rightsquigarrow_{FR}$ are defined to be the smallest evaluation-closed binary relations on closed FR configurations satisfying the rules in Figure 4.7. Let us note that $\twoheadrightarrow_{FR}$ obeys to the same rule of $\twoheadrightarrow_{HL}$. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on FR processes and configurations that satisfies the rules in Figure 3.2 and the rule E.Freeze in Figure 4.7.

Let us comment on the semantics. The communication rule is the same of HL semantics. As in roll-π a rollback begins with an application of the rule F.Start (same as the roll-π one). Then, the process of gathering all the forward history of a marked memory starts. A marked memory $[\mu; k]^\bullet$ indicates that there is an ongoing roll operation aiming to restore it. Hence, it can passes this information, via the label $\{k\}$, to all the processes directly caused by it, that is all the process tagged by a key κ such that $k > \kappa$ or $k = \kappa$. Said otherwise, a marked memory $[\mu; k]^{\bullet}$ is (k, k)-labelling (see Definition 4.9). If we consider the causal dependency relation of a marked memory as a tree rooted in the memory itself, with processes as leaves and memories as nodes we can figure out how the two rules F.InP and F.InM work. The first one models the case in which labelling information is passed to a leaf by a father memory. Note that to use this rule for the first time, when the process leaf is not frozen we need the additional structural law E.Freeze. Rule F.InM passes information from father node to a child node by labelling just the part of the configuration in the child memory that has been directly caused by it. When all the processes caused by a marked memory have been labelled, rule F.Roll can be applied. In the premises of the rule, we check that the configuration $N = \prod_{i \in I} N_i$ along with the marked memory forms a complete configuration, said otherwise the configuration N represents all the forward history of the marked memory. Then we check that all the processes in N have been labelled with the same tag of the marked memory. This is somehow similar to state that $N \blacktriangleright k$, but it is performed locally since there is no need to use a global operator such as $:>_N$ to state whether a process is interested by a $\mathsf{roll}\ k$ process: it is just sufficient to see whether it is k-labelled.

To better understand how the semantics works, let us see an example. Consider the configuration following configuration: $M = M_1 \mid (k_3 : b(Z) \triangleright Z)$, with $M_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright_{\gamma} b\langle 0\rangle \mid \mathsf{roll}\ \gamma)$. Then a possible execution can be:

$$M \twoheadrightarrow$$

(1) $\nu k.\, [M_1; k] \mid k : (b\langle 0\rangle \mid \mathsf{roll}\ k) \mid (k_3 : b(Z) \triangleright Z) \equiv$

(2) $\nu k, h_1, h_2.\, [M_1; k] \mid (\kappa_1 : b\langle 0\rangle) \mid (\kappa_2 : \mathsf{roll}\ k) \mid (k_3 : b(Z) \triangleright Z) \twoheadrightarrow$

(3) $\nu k, k_4, h_1, h_2.\, [M_1; k] \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \triangleright Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \rightsquigarrow_{\text{F.Start}}$

(4) $\nu k, k_4, h_1, h_2.\, [M_1; k]^{\bullet} \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \triangleright Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \equiv$

(5) $\nu k, k_4, h_1, h_2.\, [M_1; k]^{\bullet} \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\emptyset} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \lfloor(k_4 : 0)\rfloor_{\emptyset} \mid$
$\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\emptyset} \rightsquigarrow_{\text{F.InP}}$

(6) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_\emptyset \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \lfloor(k_4 : 0)\rfloor_\emptyset \mid$
$\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\text{F.InM}}$

(7) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \lfloor(k_4 : 0)\rfloor_\emptyset \mid$
$\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\text{F.InP}}$

(8) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \lfloor(k_4 : 0)\rfloor_{\{k\}} \mid$
$\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\text{F.Roll}}$

(9) $M_1 \mid (k_3 : b(Z) \triangleright Z) = M$

with $\kappa_1 = \langle h_1, \{h_1, h_2\}\rangle \cdot k$ and $\kappa_2 = \langle h_2, \{h_1, h_2\}\rangle \cdot k$. After the two communications, in (4) the rule F.Start is executed, causing the marking of the memory $[M_1; k]$. Since the memory $[M_1, k]^\bullet$ is (k, k)-labelling, it labels both process $\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_\emptyset$ by using the rule F.InP (6) and process $\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_\emptyset$ contained in a memory by using the rule F.InM (7). Then the continuation of the memory bearing k_4 is also frozen using the rule F.InP in (8). Once that all the processes caused by k have been labelled (frozen), then we can apply the rule F.Roll and we can get back to the initial configuration M (9). Note the use of structural law E.Freeze in (5) to label processes.

Definition 4.10 (Freeze free configuration) *A consistent configuration M is freeze free if it does not contain any frozen process (active or in memory) of the form $\lfloor\kappa : P\rfloor_S$ with $S \neq \emptyset$.*

The FR semantics allows to perform the check that all the involved processes are k-dependent in a distributed way. The proof of the correspondence between HL and FR semantics is based on the fact that freeze annotations can be removed to get the corresponding HL configuration. We start by defining the function $\gamma(\cdot)$ that removes such annotations.

Definition 4.11 *The function $\gamma(\cdot)$ from FR configurations to HL configurations is defined as follows:*

$$\begin{array}{rclrcl}
\gamma(\nu u.\, M) & = & \nu u.\, \gamma(M) & \gamma(M \mid N) & = & \gamma(M) \mid \gamma(N) \\
\gamma(\kappa : P) & = & \kappa : P & \gamma(0) & = & 0 \\
\gamma([\mu; k]) & = & [\gamma(\mu); k] & \gamma([\mu; k]^\bullet) & = & [\gamma(\mu); k]^\bullet \\
\gamma(\lfloor\kappa : P\rfloor_S) & = & \kappa : P & & &
\end{array}$$

Lemma 4.4 *For any freeze free FR configuration M, if $M \Rightarrow_{FR} N$ then $M \Rightarrow_{FR} \gamma(N) \rightsquigarrow^*_{FR} N$.*

Proof: We show that in the trace $M \Rightarrow_{FR} N$ we can move all the steps freezing processes which occur in N to the end of the trace.

The proof is by induction on the number of such freezing steps. The base case of zero steps is trivial. For the inductive case, consider the last of these freezing steps. We show that we can swap the chosen step with the following one. The thesis will follow by moving it at the end of the trace, and then applying the inductive hypothesis on the first part of the trace, which has one freezing step less. We have a case analysis on the rule used to derive the step that follows the freezing one.

F.Com: if the freezing step uses rule F.InP the thesis follows from the fact that the communication does not consume frozen processes nor memories. If the freezing step uses rule F.InM the thesis follows from the fact the communication does not consume memories.

F.Start: the thesis follows since the F.Start just adds a mark $^\bullet$, and this cannot block any freeze.

F.InP, F.InM : let us note that we have chosen the last freezing step whose effects are still present in N. This implies that if this step is followed by a F.InP or a F.InM they affect different configurations, and then the two steps can be easily swapped.

F.Roll: by hypothesis the frozen process is not removed by F.Roll since it is still present in N. Indeed, since F.Roll considers complete configurations, if it removes the labelling memory then it would remove also the frozen process, contradicting the hypothesis.

Thus we have built a trace $M \Rightarrow_{FR} N' \rightsquigarrow^*_{FR} N$. N' is freeze free by construction. Since it only differs from N because of freezes it coincides with $\gamma(N)$. □

The following proposition, similar to the Loop Theorem (Theorem 4.1), shows the semantic relation between M and $\gamma(M)$.

Proposition 4.1 *For each consistent FR configuration M, $\gamma(M) \rightsquigarrow^*_{FR} M \Rightarrow_{FR} \gamma(M)$.*

Proof: Since $\gamma(\_)$ removes just frozen processes and not marked memories, we have that M and $\gamma(M)$ differ just by frozen processes. Hence we can re-create the freezing and we obtain that $\gamma(M) \rightsquigarrow^*_{FR} M$.

We now have to show that $M \Rightarrow_{FR} \gamma(M)$. Since M is a consistent configuration then there exists a configuration M_0 such that $M_0 \Rightarrow_{FR} M$ with M_0 being freeze free and not containing any mark. We can apply Lemma 4.4 and obtain the trace $M_0 \Rightarrow_{FR} \gamma(M) \rightsquigarrow^*_{FR} M$. Now, we have two possible cases: either in $\gamma(M)$ there exists for each marked memory its corresponding roll or not. Since missing rolls can be only consumed by F.Roll applications we proceed by induction on the number of these applications. In the base case, since all the rolls are present, we have that in the reduction $M_0 \Rightarrow_{FR} \gamma(M)$ there exists some application of the rule F.Start otherwise M does not contain marked memories and we can directly apply the same technique of Theorem 4.1. If there exist F.Start applications, we can postpone all of them in order to obtain a trace of the form $M_0 \Rightarrow_{FR} M^\circ \rightsquigarrow^*_{FR} \gamma(M) \rightsquigarrow^*_{FR} M$, where M° does not contain marked memories. By executing all the ongoing rolls in M , that is by executing all the missing freeze and applying the rule F.Roll for each marked memory, we obtain the following trace $M^\circ \rightsquigarrow^*_{FR} \gamma(M) \rightsquigarrow^*_{FR} M \rightsquigarrow^*_{FR} M'$ with M' not containing marked memories. Hence we have that, $M^\circ \rightsquigarrow^*_{FR} M'$ with both M° and M' not containing marked memories. By applying the same technique of Theorem 4.1 we obtain that $M' \twoheadrightarrow^*_{FR} M^\circ$. Let us note that this Theorem works on roll-π semantics, but we can use the same arguing to have the same result in the FR semantics. Hence we have that $M \rightsquigarrow^*_{FR} M' \twoheadrightarrow^*_{FR} M^\circ \rightsquigarrow^*_{FR} \gamma(M)$, that is $M \Rightarrow_{FR} \gamma(M)$ as desired.

In the inductive case we assume that n rolls are missing. Take the first such roll. We thus have $M_0 \Rightarrow_{FR} M_1 \rightsquigarrow_{\text{F.Roll}} M_2 \Rightarrow_{FR} M$. Since M_0 is an initial process, all the memories are deleted by this roll have been created by communications in $M_0 \Rightarrow_{FR} M_1$, the memory has been marked by a F.Start in $M_0 \Rightarrow_{FR} M_1$ and all the processes frozen by steps in $M_0 \Rightarrow_{FR} M_1$. By removing all this steps does not forbid any other step. We proceed by case analysis:

F.Com: if the communication involves processes deleted by the roll application then its effect in M_2 is not present. Otherwise its effect remains in M_2.

F.InP: if this rule labels a process that will be collected by the roll, then its effect is no-more present in M_2. Otherwise it is still present since the labelled process is not collected by the roll.

F.InM: if this rule is applied on a memory that will be collected by the roll we have two cases: either it labels a process that will be collected by the roll or not. In the first case we have that its effect is no present in M_2 since the process is collected, in the second case since it labels the part of the memory not related by the roll, then this process will be still present in M_2 as an active process.

F.Start: if it marks a memory that will not be collected by the roll, then this marked memory is still present in M_2, otherwise not.

F.Roll: there is no such a step since we chose the first roll in the trace.

Thus we can build a new computation by removing all these steps and the roll. The resulting process is again M_2. Thus we have a new computation $M_0 \Rightarrow M_2 \Rightarrow M$ with one less F.Roll application. The thesis follows by inductive hypothesis.

□

We can now prove results on the operational correspondence between the two semantics. We start from the direction from HL to FR.

Proposition 4.2 (Operational correspondence: from HL to FR) *For all HL configurations M, N, if $M \rightarrow_{HL} N$ then $M \Rightarrow_{FR} N$.*

Proof: By induction on the derivation of $\rightarrow_{HL}$, with a case analysis on the used rule. The only non trivial case concerns rule H.Roll.

We have that $M = M_1 \mid [N_1; k]^\bullet$, with M a complete configuration and $M_1 \blacktriangleright k$. We have that $M \rightsquigarrow_{HL} N_1 \mid N_2$ with $N_2 = M_1 \lightning_\kappa$.

In order to have a corresponding reduction in the FR semantics we have to label all the memories and processes in M_1 with key k. Since $M_1 \blacktriangleright k$ for each such process and memory M' we have that $k :> \kappa'$ where κ' is the key of M' (for processes) or a key inside a component of M' (for memories). By definition of $:>$, there is a chain of keys $\kappa_1 > \ldots > \kappa_n$ with $k = \kappa_1$ and $\kappa' = \kappa_n$. Thanks to completeness, for all pairs (κ_i, κ_{i+1}) of keys (but possibly the last) in each chain there is a memory of the form $[\mu, \kappa_i]$ and a memory of the form $[\kappa'_i : P \mid M; \kappa_{i+1}]$ with $\kappa_i = \kappa'_i$ or $\kappa'_i = \langle \tilde{h}', h'_i \rangle \cdot \kappa'_i$. Similarly, the last element of a chain can have $\kappa'_i : P$ instead of $[\kappa'_i : P \mid M; \kappa_j]$. One can label with k all the elements by following the chains, usinng rule F.InM for labelling memories and F.InP for labelling processes. Note that if the initial set of keys is empty one can use structural congruence rule E.Freeze to label the element with an empty set of keys. When labelling has been performed it is trivial to see that applying F.Roll has the same effect of H.Roll. Note in fact that $M_1 \lightning_\kappa$ contains no labelled processes. □

We consider now the opposite direction, from FR to HL.

Proposition 4.3 (Operational correspondence: from FR to HL) *For all FR configurations M, N, if $M \rightarrow_{FR} N$ then $\gamma(M) \dashrightarrow_{HL} \gamma(N)$ (here $\dashrightarrow_{HL}$ is the reflexive closure of $\rightarrow_{HL}$).*

Proof: By induction on the derivation of $\rightarrow_{FR}$, with a case analysis on the used rule. The thesis follows by observing that the effect of rules F.InP, F.InM and E.Freeze is discarded by function $\gamma(\cdot)$, while rule F.Roll corresponds to rule H.Roll. □

We can use the two results above to ensure that HL and FR semantics are equivalent with respect to weak barbed congruence.

Theorem 4.3 (Behavioral correspondence: HL vs FR) *For each (consistent) HL configuration M we have $M\ {}_{HL}\overset{.}{\approx}{}^{c}_{FR}\ M$.*

Proof: We show that the relation:

$$_{HL}\mathcal{R}_{FR} = \left\{ (\mathbb{C}[\gamma(M)], \mathbb{C}[M]) \middle| \begin{array}{c} M \text{ is a consistent FR configuration } \wedge \\ \mathbb{C}[\gamma(M)], \mathbb{C}[M] \text{ are consistent} \end{array} \right\}$$

is a weak barbed bisimulation with respect to HL and FR.

Observe that $\mathbb{C}[\gamma(M)] = \gamma(\mathbb{C}[M])$, thus it is enough to consider pairs of consistent configurations of the form $(\gamma(M), M)$.

For the condition on barbs, first observe that if M has a barb, then $\gamma(M)$ has the same barb. For the opposite direction, if $\gamma(M)$ has a barb, using Proposition 4.1 we have that $M \Rightarrow \gamma(M)$, thus M has the same (weak) barb.

For the condition on reductions we have a case analysis according to the type of reduction and the configuration that makes the challenge. Challenges from FR configuration M are matched thanks to Proposition 4.3. Challenges from HL configuration $\gamma(M)$ are a bit more tricky. For backward reductions, we can use Proposition 4.2, while for forward reductions if one of the processes involved (message or trigger) is frozen in M (and hence cannot communicate) then one has to apply Proposition 4.1 to have $M \Rightarrow \gamma(M)$ and then match the communication, otherwise if both processes are not frozen then the communication can be directly matched. □

4.4.3 Roll Semantics

In the second refinement, we rely on asynchronous notification to freeze threads, breaking down the synchrony required by rules F.InP and F.InM (in Figure 4.7): instead of using the predicate (k, k_1)-labelling we use *runtime* roll notifications, noted as rl, to notify processes that are part of a rollback operation. Figure 4.8 shows the syntax of the *RL* (for roll) refinement, where the main change with respect to the previous one is that now configurations include roll notifications of the form $\mathsf{rl}\ k, \kappa$. A notification of the form $\mathsf{rl}\ k, \kappa$ means that there is an ongoing rollback targeting the memory bearing the tag k, and that the thread tagged by κ should be notified of being part of the rollback aiming to restore k.

$$\begin{array}{rl} P, Q ::= & \ldots \\ M, N ::= & \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid \kappa : P \mid \lfloor \kappa : P \rfloor_S \mid [\mu; k] \mid [\mu; k]^{\bullet} \\ & \mid \mathsf{rl}\ k, \kappa \end{array}$$

Figure 4.8: Syntax of RL refinement.

The RL semantics $\rightarrow_{RL}$ of roll-π is defined as for the HL one (cf. Section 4.2.1), as $\rightarrow_{RL} = \twoheadrightarrow_{RL} \cup \rightsquigarrow_{RL}$, where relations $\twoheadrightarrow_{RL}$ and $\rightsquigarrow_{RL}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 4.9. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on RL processes and configurations that satisfies the rules in Figure 3.2 plus rules: E.Freeze (in Figure 4.7) and E.GB (in Figure 4.9).

Figure 4.9 depicts the semantics rules and additional structural law used by the RL refinement with respect to the FR refinement. The communication rule (not shown here) is the same as that of roll-π. A rollback operation begins with the application of the rule R.Start. This rule has two side effects: it marks the memory (as the previous semantics) and it generates a *runtime notification* of the form $\mathsf{rl}\ k, k$ in order to freeze its descendants. Rule Span is used to propagate the notification through the causal dependency tree, that is if there is a notification of the form $\mathsf{rl}\ k, \kappa$ and the thread tagged by κ is contained in a memory (is part of a communication) then the process identified by κ gets frozen and a notification to freeze the memory descendants is generated. Rule R.Branch just *disseminates* a notification of the form $\mathsf{rl}\ k, k_1$ through a parallel composition. This rule is applied in the case in which the process bearing tag k_1 was a parallel composition, and an application of the structural law E.TagP (see Figure 4.4) splits it into several threads. Hence, a notification is created for each single thread in which the process has been split. Therefore, rules R.Branch and R.Span serve to disseminate notifications through the tree that the causal relation forms. Rule R.Up adds the information about a roll process to a thread process. Note that, as for the FR semantics, in this semantics it is important the use of the rule E.Freeze (see Figure 4.7), since it allows to consider a thread as a frozen process labelled by the empty set. Structural law E.GB is used to collect rl notifications that are no more needed, that is notifications on memories that have been already deleted. Indeed, if a memory is deleted

$$\text{(R.S{\scriptsize TART})}\ (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{RL} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^{\bullet} \mid \mathsf{rl}\ k, k$$

$$\text{(R.S{\scriptsize PAN})}\quad \frac{k \notin S}{\mathsf{rl}\ k, \kappa_1 \mid [\lfloor \kappa_1 : P \rfloor_S \mid M; k_2]^{\circ} \rightsquigarrow_{RL} [\lfloor \kappa_1 : P \rfloor_{S \cup \{k\}} \mid M; k_2]^{\circ} \mid \mathsf{rl}\ k, k_2}$$

$$\text{(R.B{\scriptsize RANCH})}\quad \frac{M = \langle h_i, \tilde{h} \rangle \cdot k_1 : P \qquad \vee \qquad M = [\langle h_i, \tilde{h} \rangle \cdot k_1 : P \mid N; k_2]}{\mathsf{rl}\ k, k_1 \mid M \rightsquigarrow_{RL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ k, \langle h_i, \tilde{h} \rangle \cdot k_1 \mid M}$$

$$\text{(R.U{\scriptsize P})}\quad \frac{k \notin S}{\mathsf{rl}\ k, \kappa_1 \mid \lfloor \kappa_1 : P \rfloor_S \rightsquigarrow_{RL} \lfloor \kappa_1 : P \rfloor_{S \cup \{k\}}}$$

$$\text{(R.R{\scriptsize OLL})}\quad \frac{\mathtt{complete}([\mu; k] \mid \prod_{i \in I} N_i) \quad \forall i \in I. N_i \text{ is } k\text{-labelled}}{[\mu; k]^{\bullet} \mid \prod_{i \in I} N_i \rightsquigarrow_{RL} \mu \mid (\prod_{i \in I} N_i)\lightning_k}$$

$$\text{(E.GB)}\quad \nu l. \prod_{i \in I} \mathsf{rl}\ k_i, l \mid \prod_{j \in J} \mathsf{rl}\ k_j, \langle h_j, \tilde{h}_j \rangle \cdot l \equiv_{RL} 0$$

Figure 4.9: RL semantics.

then all the rl notifications aiming to collect its descendants are meaningless and does not have any effect on the entire configuration. Therefore, they can be garbage collected (considered as 0).

To better understand how the semantics works, let us see an example (the same as in Section 4.4.2). Consider the configuration $M = M_1 \mid (k_3 : b(Z) \triangleright Z)$, with $M_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright_\gamma b\langle 0\rangle \mid \mathsf{roll}\ \gamma)$. Then a possible execution can be:

$$\begin{array}{ll}
 & M \twoheadrightarrow \\
(1) & \nu k.\, [M_1; k] \mid k : (b\langle 0\rangle \mid \mathsf{roll}\ k) \mid (k_3 : b(Z) \triangleright Z) \equiv \\
(2) & \nu k, h_1, h_2.\, [M_1; k] \mid (\kappa_1 : b\langle 0\rangle) \mid (\kappa_2 : \mathsf{roll}\ k) \mid (k_3 : b(Z) \triangleright Z) \twoheadrightarrow \\
(3) & \nu k, k_4, h_1, h_2.\, [M_1; k] \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \triangleright Z); k_4] \mid (k_4 : 0) \mid \\
 & (\kappa_2 : \mathsf{roll}\ k) \rightsquigarrow_{\text{R.S}\scriptsize{\text{TART}}} \\
(4) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \mathsf{rl}\ k, k \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \triangleright Z); k_4] \mid (k_4 : 0) \mid \\
 & (\kappa_2 : \mathsf{roll}\ k) \rightsquigarrow_{\text{R.B}\scriptsize{\text{RANCH}}} \\
(5) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \mathsf{rl}\ k, \kappa_1 \mid \mathsf{rl}\ k, \kappa_2 \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & (k_4 : 0) \mid (\kappa_2 : \mathsf{roll}\ k) \equiv \\
(6) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \mathsf{rl}\ k, \kappa_1 \mid \mathsf{rl}\ k, \kappa_2 \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_\emptyset \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & \lfloor(k_4 : 0)\rfloor_\emptyset \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_\emptyset \rightsquigarrow_{\text{R.S}\scriptsize{\text{PAN}}} \\
(7) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \mathsf{rl}\ k, \kappa_2 \mid \mathsf{rl}\ k, k_4 \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & \lfloor(k_4 : 0)\rfloor_\emptyset \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_\emptyset \rightsquigarrow_{\text{R.U}\scriptsize{\text{P}}} \\
(8) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \mathsf{rl}\ k, k_4 \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & \lfloor(k_4 : 0)\rfloor_\emptyset \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\text{R.U}\scriptsize{\text{P}}} \\
(9) & \nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & \lfloor(k_4 : 0)\rfloor_{\{k\}} \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\text{R.R}\scriptsize{\text{OLL}}} \\
(10) & M
\end{array}$$

with $\kappa_1 = \langle h_1, \{h_1, h_2\}\rangle \cdot k$ and $\kappa_2 = \langle h_2, \{h_1, h_2\}\rangle \cdot k$. Starting from M after two communications (3), the rule R.Start is applied on the memory $[M_1, k]$ (4) causing its marking and the generation of the runtime notification $\mathsf{rl}\ k, k$. Since the process bearing the tag k has been split into two primitive threads identified by tags κ_1 and κ_2, then the notification $\mathsf{rl}\ k, k$ is split into the two *sub-notifications* $\mathsf{rl}\ k, \kappa_1$ and $\mathsf{rl}\ k, \kappa_2$ using the rule R.Branch (5). Process tagged by κ_1 is part of a memory, then the rule R.Span is applied (7), with the effect of freezing the process tagged by κ_1, inside the memory, and generating a notification for the descendants of the memory containing it, that is $\mathsf{rl}\ k, k_4$. In (8) the rule R.UP is applied on the process tagged by κ_2, while the same rule in (9) is applied on the process tagged by k_3. Now, since there are no more notifications and all the processes generated *because of* k have been frozen (and labelled by k) the rule R.ROLL can be applied and the whole configuration can get back to the initial one (M). Note again in (5) the use of structural law E.FREEZE.

As already pointed out, the difference with the FR refinement is just the use of asynchronous notifications to freeze processes, instead of using the predicate (k, k_1)-labelling. Hence, rl notifications can be seen as an annotation in the semantics, and we define a function $\gamma_r(\cdot)$ removing them.

Definition 4.12 *The function $\gamma_r(\cdot)$ from RL configurations to FR configurations is defined as follows:*

$$\begin{aligned}
\gamma_r(\nu u.\, M) =& \nu u.\, \gamma_r(M) & \gamma_r(M \mid N) =& \gamma_r(M) \mid \gamma_r(N)\\
\gamma_r(\kappa : P) =& \kappa : P & \gamma_r(0) =& 0\\
\gamma_r([M; k]) =& [M; k] & \gamma_r([M; k]^\circ) =& [M; k]^\circ\\
\gamma_r(\lfloor \kappa : P \rfloor_S) =& \lfloor \kappa : P \rfloor_S & \gamma_r(\mathsf{rl}\ k, \kappa_1) =& 0
\end{aligned}$$

The correspondence is based on the following invariant, which defines the behavior of rl notifications.

Lemma 4.5 *For each RL configuration $M \equiv_{RL} \nu \tilde{n}.\, [\mu, k_1]^\circ \mid N$ we have that $[\mu, k_1]^\circ$ is (k, k_1)-labeling if and only if:*

- *either there is in N a process $\lfloor k_1 : P \rfloor_S$ or a memory $[\lfloor k_1 : P \rfloor_S \mid N'; k_2]^\circ$ with $k \in S$;*
- *or there is in N a configuration of the form $\mathsf{rl}\ k, k_1$;*
- *or there exists $\tilde{h}$ and $\langle h_i, \tilde{h} \rangle \cdot k_1$ is free in N and for each $h_i \in \tilde{h}$:*
  - *either there is in N a configuration $\lfloor \langle h_i, \tilde{h} \rangle \cdot k_1 : P \rfloor_S$ or memory $[\lfloor \langle h_i, \tilde{h} \rangle \cdot k_1 : P \rfloor_S \mid N'; k_3]^\circ$ with $k \in S$;*
  - *or there is in N a configuration of the form $\mathsf{rl}\ k, \langle h_i, \tilde{h} \rangle \cdot k_1$.*

Proof: The proof is by induction on the number of reductions from the initial configuration to M, with a case analysis on the used axiom. For the $\Rightarrow$ case we have the following cases:

Com: the only memory created by the reduction is not (k, k_1)-labelling for any (k, k_1), thus memory $[\mu, k]^\circ$ should be in the context (up to structural congruence a reduction is an application of an axiom, Com in this case, in a suitable context). By inductive hypothesis the labelled processes or memories or the roll configurations existed before the reduction. Since the reduction can not delete any of them (neither remove a complex key), then the thesis holds also after the reduction.

R.Start: if the memory $[\mu, k_1]^\circ$ is the one tagged by the rule, than the thesis holds directly since the rule also creates a configuration of the form $\mathsf{rl}\ k_1, k_1$. Otherwise it holds by inductive hypothesis.

R.Span: if the memory $[\mu, k_1]^\circ$ is the one appearing in the rule, then the thesis holds trivially since there is a corresponding configuration $\mathsf{rl}\ \kappa, k_1$. Otherwise if the memory $[\mu, k_1]^\circ$ is in the context, then we can apply inductive hypothesis to know that the condition was satisfied before the reduction. If the configuration satisfying the condition was not the existence of $\mathsf{rl}\ k, k_1$ then the condition is still satisfied for the same reason after the reduction. Otherwise the condition becomes satisfied since a memory $[\lfloor \kappa_1 : P \rfloor_{S \cup \{k\}} \mid N; k_2]$ is created with $\kappa_1 = k$ or $\kappa_1 = \langle \tilde{h}, h_i \rangle \cdot k_1$.

R.Branch: the memory $[\mu, \kappa_1]^\circ$ is necessarily in the context or in the term M. Since M is preserved the same reasoning can be done in the two cases. If the reason for the condition to be satisfied is not the existence of $\mathsf{rl}\ k, k_1$, then the thesis follows trivially by inductive hypothesis. Otherwise it follows trivially since there is $\tilde{h}$ such that $\langle \tilde{h}, h_i \rangle \cdot k_1$ occurs in the term and for each $h_i \in \tilde{h}$ there is a configuration of the form $\mathsf{rl}\ k, \langle \tilde{h}, h_i \rangle \cdot k_1$.

R.Up: the memory $[\mu, k_1]^\circ$ is necessarily in the context. If the reason for the condition to be satisfied is not the existence of $\mathsf{rl}\ k, k_1$, then the thesis follows trivially by inductive hypothesis. Otherwise it follows trivially since there is a process $\lfloor \kappa_1 : P \rfloor_S$ with $k \in S$ and $\kappa_1 = k$ or $\kappa_1 = \langle \tilde{h}, h_i \rangle \cdot k_1$.

R.Roll: there are no memories in the right hand side of the axiom, thus the (k, k_1)-labelling memory $[\mu, k_1]^\circ$ is necessarily in the context. Let us consider the different possibilities the condition could have been satisfied before the reduction. If it is because of the existence of a rl notification, then it is still satisfied after the reduction since no such term is removed (and complex keys are removed only if the corresponding memory is removed). If it is because of the existence of a process or memory not removed by the reduction then the thesis is still satisfied after the reduction. If it is satisfied because of memory $[\mu; k_2]^\bullet$ (the target memory) then one of its internal processes should be labelled by k, and thus the term μ in the result makes the condition satisfied. Assume now that the condition was satisfied thanks to some process or memory N_i. Let us consider the labelled process (possibly inside the memory). If the labelling memory was k'-labelled where k'' is the key to be rolled-back then it disappeared and thus nothing has to be proved. Otherwise the only possibility is that N_i was a memory with a subprocess k'-labelled and a subprocess k-labelled (but not k'-labelled). Then the subprocess k-labelled occurs in $(\prod_{i\in I} N_i)\lightning_{\kappa'}$, satisfying the thesis.

In the case $\Leftarrow$ we prove that if $M \equiv_{RL} \nu\tilde{n}.\,\mathsf{rl}\ k, \kappa_1 \mid N$ with $\mathsf{rl}\ k, \kappa_1$ non collectable via $\equiv_{RL}$, then there exists in N a memory $[\mu, k_1]^\circ$ that is (k, k_1)-labelling with $\kappa_1 = k_1$ or $\kappa_1 = \langle\tilde{h}, h_i\rangle \cdot k_1$. As the above case we proceed by case analysis on the applied axiom. All the cases are simple. □

We can now prove the correspondence theorem.

Theorem 4.4 (Behavioral correspondence: FR vs RL) *For each (consistent) FR configuration M we have $M\ {}_{FR}\overset{\cdot\ c}{\approx}_{RL}\ M$.*

Proof: We show that the relation:

$$_{FR}\mathcal{R}_{RL} = \left\{ (\mathbb{C}[\gamma_r(M)], \mathbb{C}[M]) \middle| \begin{array}{c} M \text{ is a consistent RL configuration} \wedge \\ \mathbb{C}[\gamma_r(M)], \mathbb{C}[M] \text{ are consistent} \end{array} \right\}$$

is a weak barbed bisimulation with respect to FR and RL.

The proof for barbs follows trivially since rl notifications have no barb. The proof for reductions is trivial for reductions derived using rules R.Comm, R.Start, and R.Roll and the corresponding FR rules. Rules F.InP and F.InM are matched respectively by rules R.Up and R.Span since by Lemma 4.5 if there is a memory (k, k_1)-labelling then there exists a corresponding rl notification. On the other side, rules R.Span and R.Up are matched by rules F.InP and F.InM by using again Lemma 4.5 since if there exists a notification $\mathsf{rl}\ k, \kappa_1$ then there exists a memory that is (k, k_1)-labelling, with $\kappa_1 = k_1$ or $\kappa_1 = \langle\tilde{h}, h_i\rangle \cdot k_1$. Rule R.Branch is matched by the identity, since is does not affect either memories or threads and all the rl notifications are deleted by $\gamma(\_)$. □

4.4.4 Distributed Semantics

We now introduce the last refinement semantics. With the previous semantics we have broken down the atomicity of gathering all the processes related to a particular tag, that is the forward history of a target memory. But restoring the content of a memory and deleting its effects is still an atomic step involving an unbounded number of processes. With this semantics, that we call DS (for distributed), we mimic the backward reduction of $\rho\pi$, that is we exploit memories to get back of one step (of communication) at time. In this way we break down the atomicity of the rule H.Roll. This time there is no change of the syntax of RL.

$$\text{(D.Stop)} \quad \frac{S \neq \emptyset}{[\mu; k]^{\circ} \mid \lfloor k : P \rfloor_S \rightsquigarrow_{DS} \mu}$$

$$\text{(E.TagPFrS)} \quad \frac{\tilde{h} = \{h_1, \ldots, h_n\} \qquad n \geq 2}{\lfloor k : \prod_{i=1}^{n} \tau_i \rfloor_S \equiv_{DS} \nu \tilde{h}.\ \prod_{i=1}^{n} \lfloor (\langle h_i, \tilde{h} \rangle \cdot k : \tau_i) \rfloor_S}$$

Figure 4.10: DS semantics.

The DS semantics $\rightarrow_{DS}$ of roll-π is defined as for the HL one (cf. Section 4.2.1), as $\rightarrow_{DS} = \twoheadrightarrow_{DS} \cup \rightsquigarrow_{DS}$, where relations $\twoheadrightarrow_{DS}$ and $\rightsquigarrow_{DS}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 4.9 (except for R.Roll) and Figure 4.10. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on DS processes and configurations that satisfies the rules in Figure 3.2 plus rules: E.Freeze (in Figure 4.7) and E.GB (in Figure 4.9) and rule in Figure 4.10. Figure 4.10 depicts the extra semantic rules and structural laws used by the DS refinement. The main change with respect the RL semantics is that we replace the rule R.Roll with the rule D.Stop. This rule simply restores the content of a memory (that may contain frozen processes) if its descendant is a frozen process. As already said, the rollback operation is divided into two phases: (i) gathering all the processes related to a particular memory and then (ii) deleting the effects of this memory and restoring its content. So, once we have frozen all the processes interested by a roll operation, instead of deleting them at once, we can undo all the memories step by step (as in $\rho\pi$), starting from the most recent memories, that is memories whose continuations represent leaves of the causal tree. Let us note that, in this semantics, labelling information are meaningless since a rollback is done by several steps. The structural law E.TagPFrS is similar to the roll-π (and $\rho\pi$) rule E.TagP, and it is quite important since it allows to build back a frozen parallel composition from a parallel composition of frozen threads belonging to the same memory. Therefore, it is always used (if necessary) before an application of the rule D.Stop, since to be applied D.Stop it requires that a process bearing the memory tag has to be frozen.

As in the other sections, let us consider an example to better understand how the DS semantics works. Consider the configuration $M = M_1 \mid (k_3 : b(Z) \triangleright Z)$, with $M_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright_\gamma b\langle 0\rangle \mid \mathsf{roll}\ \gamma)$. Then, following the example of execution of the previous section we have:

$$\begin{array}{ll}
 & M \Rightarrow \\
(9) & \nu k, k_4, h_1, h_2. [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z); k_4] \mid \\
 & \lfloor(k_4 : 0)\rfloor_{\{k\}} \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \rightsquigarrow_{\textsc{D.Stop}} \\
(10) & \nu k, h_1, h_2. [M_1; k]^\bullet \mid \lfloor(\kappa_1 : b\langle 0\rangle)\rfloor_{\{k\}} \mid (k_3 : b(Z) \triangleright Z) \\
 & \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor_{\{k\}} \equiv \\
(11) & \nu k. [M_1; k]^\bullet \mid (k_3 : b(Z) \triangleright Z) \mid \lfloor k : (\mathsf{roll}\ k \mid b\langle 0\rangle)\rfloor_{\{k\}} \rightsquigarrow_{\textsc{D.Stop}} \\
(12) & M_1 \mid (k_3 : b(Z) \triangleright Z) = M
\end{array}$$

with $\kappa_1 = \langle h_1, \{h_1, h_2\}\rangle \cdot k$ and $\kappa_2 = \langle h_2, \{h_1, h_2\}\rangle \cdot k$. Following the reductions of the previous section from M we reach the configuration in (9), where all the processes caused by k have been frozen (and labelled). Now, instead of executing the rollback in one atomic step (as in the RL semantics) the DS semantics undoes forward steps (memories) one by one. In (9) there is a memory bearing k_4 and there is a frozen process tagged by k_4, that is the continuation of the memory bearing the tag k_4 is frozen. Hence, rule D.Stop can be applied and the content of the memory is restored and its continuation deleted (10). Now by applying the structural law E.TagPFrS on the two frozen processes tagged respectively by κ_1 and κ_2, the process $\lfloor k : (\mathsf{roll}\ k \mid b\langle 0\rangle)\rfloor_{\{k\}}$ can be retrieved (11). By applying again the rule D.Stop the configuration gets back to the initial one, as desired.

We prove now results on the operational correspondence between the two semantics. We start from the direction from RL to DS.

Proposition 4.4 (Operational correspondence: from RL to DS) *For all RL configurations M, N, if $M \rightarrow_{RL} N$ then $M \Rightarrow_{DS} N$.*

Proof: The proof is by case analysis on the axiom used to derive the transition. All the cases are trivial but for rule HL-Roll. In this case the proof is by induction on the number of components in the term. We have that in M there exist a configuration $[\mu, k]^\bullet \mid \prod_{i \in I} N_i$ such that $\forall i \in I. N_i \blacktriangleright k$ and $\texttt{complete}([\mu, k]^\bullet \mid \prod_{i \in I} N_i)$. Thanks to completeness and since all the processes are k-labelled, we know that there are chains connecting them to the memory bearing k. We apply rule D.Stop recursively to all the leaves, using the structural congruence E.TagPFr to join brother leaves before rolling-back. Processes will completely disappear as required. For memories, at least one of the two internal processes is k-labelled and thus disappears when it interacts with the father memory. Processes which are not k-labelled instead are preserved, as expected since they are in $(\prod_{i \in I} N_i)\lightning_{\kappa}$. □

We need a lemma stating that DS reductions derived using rule D.Stop commute with other reductions. To help the notation, we write $M \hookrightarrow_{DS} N$ if $M \rightsquigarrow_{DS} N$ and the reduction is derived using rule D.Stop. Also, $\hookrightarrow^*_{DS}$ is the reflexive and transitive closure of $\hookrightarrow_{DS}$.

Lemma 4.6 (Stop swap) *For each consistent DS configuration N, if $N \hookrightarrow_{DS} M$ and $N \rightarrow_{DS} N'$ then there is a DS configuration M' such that $M \dashrightarrow_{DS} M'$ and $N' \hookrightarrow_{DS} M'$ (here $\dashrightarrow_{DS}$ is the reflexive closure of $\rightarrow_{DS}$).*

Proof: The proof is by case analysis on the rule used to derive $M \rightarrow_{DS} M'$.

R.Com: the premises of the two rules are necessarily disjoint, thus they trivially commute.

R.Start: the only difficult case is when the memory in the premise of R.Start is also the memory in the premise of D.Stop, since in all the other cases the two reductions easily commute. By construction the configuration $\kappa_1 : \mathsf{roll}\ \kappa$ exists only if $\kappa :> \kappa_1$. However the existence of a configuration $\lfloor \kappa : P \rfloor_S$ implies that there are no other descendants, thus this case can never happen.

R.Span: Again the only difficult case is when the memory in rule R.Span is also the memory in rule D.Stop. Thus we have a computation of the form:

$$\begin{array}{l} \mathsf{roll}\ \kappa, \kappa_1 \mid [\lfloor \kappa_1 : P \rfloor_S \mid M; k_2]^\circ \mid \lfloor k_2 : Q \rfloor_{S'} \rightarrow \\ \qquad [\lfloor \kappa_1 : P \rfloor_{S \cup \{\kappa\}} \mid M; k_2]^\circ \mid \mathsf{roll}\ \kappa, k_2 \mid \lfloor k_2 : Q \rfloor_{S'} \hookrightarrow \\ \qquad\qquad \lfloor \kappa_1 : P \rfloor_{S \cup \{\kappa\}} \mid M \mid \mathsf{roll}\ \kappa, k_2 \end{array}$$

We also have a computation:

$$\begin{aligned}\mathsf{roll}\ \kappa, \kappa_1 \mid [\lfloor \kappa_1 : P \rfloor_S \mid M; k_2]^\circ \mid \lfloor k_2 : Q \rfloor_{S'} \hookrightarrow \\ \mathsf{roll}\ \kappa, \kappa_1 \mid \lfloor \kappa_1 : P \rfloor_S \mid M \to \\ \lfloor \kappa_1 : P \rfloor_{S \cup \{\kappa\}} \mid M\end{aligned}$$

where the last reduction is derived using rule R.Up. Since in the result of the computation the process on k_2 has been removed, then there are no other processes on k_2 and $\mathsf{roll}\ \kappa, k_2$ can be garbage collected using rule E.Gb.

R.Branch: the two rules commute easily.

R.Up: the difficult case is when the frozen process in the two rules is the same. After R.Up, D.Stop can still be executed. After D.Stop, no step is required, and $\mathsf{roll}\ \kappa, k_1$ can be garbage collected since k_1 has disappeared.

D.Stop: two stop rules necessarily act on disjoint configurations since there are at most one memory and one process with the same key. Thus they trivially commute.

□

We can now prove the equivalence result.

Theorem 4.5 (Behavioral correspondence: RL vs DS) *For each (consistent) RL configuration M we have $M\ {}_{RL}\overset{\cdot}{\approx}^{c}_{DS}\ M$.*

Proof: We have to show that for each consistent RL configuration M and each $\mathbb{C}[\cdot]$ such that $\mathbb{C}[M]$ is consistent, $\mathbb{C}[M]\ {}_{RL}\approx_{DS}\ \mathbb{C}[M]$. To this end we show that the relation:

$${}_{RL}\mathcal{R}_{DS} = \left\{ (M,N) \left| \begin{array}{c} M \text{ is a consistent RL configuration} \wedge \\ N \text{ is a consistent DS configuration} \wedge \\ N \hookrightarrow^*_{DS} M \end{array} \right. \right\}$$

is a weak barbed bisimulation with respect to RL and DS.

Challenges from M (both barbs and reductions) are trivially answered since $N \hookrightarrow^*_{DS} M$. The only tricky case is when the challenge is a reduction derived using rule R.Roll, but this can be answered thanks to Proposition 4.4.

Let us consider challenges from N. If N has a barb then M has the same barb since rule D.Stop never removes barbs. If $N \rightarrow_{DS} N'$ then using multiple times Lemma 4.6 one can derive $N \hookrightarrow^*_{DS} M \dashrightarrow_{DS} M'$ and $N' \hookrightarrow^*_{DS} M'$. If $M \dashrightarrow_{DS} M'$ does not use rule D.Stop then $M \dashrightarrow_{RL} M'$ and the thesis follows. Otherwise $N \hookrightarrow^*_{DS} M'$ and the pair (M', N) is in the relation as desired. □

4.5 A low level algorithm for roll-π

The semantics defined in Section 4.2.1 (see Figure 4.5) captures the behavior of rollback, but it relies on global checks on large parts of the configuration, for verifying that it is complete and κ-dependent. This makes it difficult to implement directly such a semantics, even more so in a distributed setting.

We present now a low-level (written LL) semantics, where the conditions above are verified incrementally by relying on communication of rl notifications. We show that the LL semantics captures the same intuition as the one introduced in Section 4.2.1 by proving that, given a (consistent) configuration, its behavior under the two semantics are weak barbed congruent according to Definition 4.7.

To avoid confusion between the two semantics (and others semantics used in the proof), we use a subscript LL to identify all the elements (reductions, structural congruence, ...) referred to the low-level semantics presented here, and HL (for high-level) for the semantics described in Section 4.2.1.

The LL semantics $\rightarrow_{LL}$ of roll-π is defined as the HL one (cf. Section 4.2.1), as $\rightarrow_{LL} = \twoheadrightarrow_{LL} \cup \rightsquigarrow_{LL}$, where relations $\twoheadrightarrow_{LL}$ and $\rightsquigarrow_{LL}$ are defined to be the smallest evaluation-closed binary relations on closed LL configurations satisfying the rules in Figure 4.11. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on LL processes and configurations that satisfies the rules in Figure 3.2 and in Figure 4.12.

LL configurations differ from HL configurations in two aspects. First, tagged processes (inside or outside memories) can be frozen, denoted $\lfloor \kappa : P \rfloor$, to indicate that they are participating to a rollback (rollback is no longer atomic). Second, LL configurations include notifications of the form rl κ, used to notify a tagged process with key κ to enter a rollback.

Let us describe the LL rules. Communication rule L.Com is as in HL. The main idea for rollback is that when a memory pointed by a roll is marked (rule L.Start), a notification rl k is generated. This notification is propagated by rules L.Span and L.Branch. Rule L.Span also freezes threads inside memories, specifying that they will be eventually removed by the rollback. Rule L.Branch (where the predicate "κ occurs in M" means that either $M = \kappa : P$ or $M = [\mu; k']^{\circ}$ with $\kappa : P \in M$) is used when the target configuration has been split into multiple threads: a notification has to be sent to each of them. Rule L.Up is similar to L.Span, but it applies to tagged processes outside memories. It also stops the propagation of the rl notification. The main idea is that by using rules L.Span, L.Branch, and L.Up one is able to tag all the causal descendants of a marked memory. Finally, rule L.Stop rolls-back a single computation step by removing a frozen process and freeing the content of the memory created with it. In the LL semantics a rollback request is thus executed incrementally, while it was atomic in the HL semantics (rule H.Roll). The LL semantics also exploits an extended structural congruence. The axiom E.Gb is used to garbage collect rl notifications on keys that are no more used in the entire configuration. The operator $\subset$ on keys is defined as follows: $k \subset \kappa$ if $\kappa = k$ or $\kappa = \langle h_i, \tilde{h}\rangle \cdot k$. Axiom E.TagPFr is an adaptation of the axiom E.TagP to deal with frozen processes.

$$\text{(L.Com)} \quad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow_{LL} \nu k.\, (k : Q\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(L.Start)} \quad (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k] \rightsquigarrow_{LL} (\kappa_1 : \mathsf{roll}\ k) \mid [\mu; k]^{\bullet} \mid \mathsf{rl}\ k$$

$$\text{(L.Span)} \quad \mathsf{rl}\ \kappa_1 \mid [\kappa_1 : P \mid M; k]^{\circ} \rightsquigarrow_{LL} [\lfloor \kappa_1 : P \rfloor \mid M; k]^{\circ} \mid \mathsf{rl}\ k$$

$$\text{(L.Branch)} \quad \frac{\langle h_i, \tilde{h}\rangle \cdot k \text{ occurs in } M}{\mathsf{rl}\ k \mid M \rightsquigarrow_{LL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \mid M}$$

$$\text{(L.Up)} \quad \mathsf{rl}\ \kappa_1 \mid (\kappa_1 : P) \rightsquigarrow_{LL} \lfloor \kappa_1 : P \rfloor \qquad \text{(L.Stop)} \quad [\mu; k]^{\circ} \mid \lfloor k : P \rfloor \rightsquigarrow_{LL} \mu$$

Figure 4.11: Reduction rules for LL.

We now show an example to clarify the semantics. Consider the configuration $M = M_1 \mid (k_3 : b(Z) \triangleright Z)$, with $M_1 = (k_1 : a\langle 0\rangle) \mid (k_2 : a(X) \triangleright_\gamma b\langle 0\rangle \mid$

$$\text{(E.G{\scriptsize B})} \quad \nu k.\ \prod_{i\in I} \mathsf{rl}\ \kappa_i \equiv_{LL} 0 \quad k \subset \kappa_i$$

$$\text{(E.T{\scriptsize AG}PF{\scriptsize R})} \quad \frac{\tilde{h} = \{h_1, \ldots, h_n\} \qquad n \geq 2}{\lfloor k : \prod_{i=1}^{n} \tau_i \rfloor \equiv_{LL} \nu \tilde{h}.\ \prod_{i=1}^{n} \lfloor \langle h_i, \tilde{h} \rangle \cdot k : \tau_i \rfloor}$$

Figure 4.12: Additional structural laws for LL.

$\mathsf{roll}\ \gamma$). Then a possible execution can be:

$M \twoheadrightarrow$

(1) $\nu k.\, [M_1; k] \mid k : (b\langle 0\rangle \mid \mathsf{roll}\ k) \mid (k_3 : b(Z) \rhd Z) \equiv$

(2) $\nu k, h_1, h_2.\, [M_1; k] \mid (\kappa_1 : b\langle 0\rangle) \mid (\kappa_2 : \mathsf{roll}\ k) \mid (k_3 : b(Z) \rhd Z) \twoheadrightarrow$

(3) $\nu k, k_4, h_1, h_2.\, [M_1; k] \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \rhd Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \rightsquigarrow_{\text{L.S{\scriptsize TART}}}$

(4) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \rhd Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \mid \mathsf{rl}\ k \rightsquigarrow_{\text{L.B{\scriptsize RANCH}}}$

(5) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [(\kappa_1 : b\langle 0\rangle) \mid (k_3 : b(Z) \rhd Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \mid \mathsf{rl}\ \kappa_1 \mid \mathsf{rl}\ \kappa_2 \rightsquigarrow_{\text{L.S{\scriptsize PAN}}}$

(6) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor \mid (k_3 : b(Z) \rhd Z); k_4] \mid (k_4 : 0) \mid$
$(\kappa_2 : \mathsf{roll}\ k) \mid \mathsf{rl}\ k_4 \mid \mathsf{rl}\ \kappa_2 \rightsquigarrow_{\text{L.U{\scriptsize P}}}$

(7) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor \mid (k_3 : b(Z) \rhd Z); k_4] \mid \lfloor(k_4 : 0)\rfloor \mid$
$(\kappa_2 : \mathsf{roll}\ k) \mid \mathsf{rl}\ \kappa_2 \rightsquigarrow_{\text{L.U{\scriptsize P}}}$

(8) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid [\lfloor(\kappa_1 : b\langle 0\rangle)\rfloor \mid (k_3 : b(Z) \rhd Z); k_4] \mid \lfloor(k_4 : 0)\rfloor \mid$
$\lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor \rightsquigarrow_{\text{L.S{\scriptsize TOP}}}$

(9) $\nu k, k_4, h_1, h_2.\, [M_1; k]^\bullet \mid \lfloor(\kappa_1 : b\langle 0\rangle)\rfloor \mid (k_3 : b(Z) \rhd Z) \mid \lfloor(\kappa_2 : \mathsf{roll}\ k)\rfloor \equiv$

(10) $\nu k.\, [M_1; k]^\bullet \mid (k_3 : b(Z) \rhd Z) \mid \lfloor k : (b\langle 0\rangle \mid \mathsf{roll}\ k)\rfloor \rightsquigarrow_{\text{L.S{\scriptsize TOP}}}$

(11) $M_1 \mid (k_3 : b(Z) \rhd Z)$

with $\kappa_1 = \langle h_1, \{h_1, h_2\}\rangle \cdot k$ and $\kappa_2 = \langle h_2, \{h_1, h_2\}\rangle \cdot k$.

One can see that the rollback operation starts with the application of the rule L.Start (3), whose effects are (i) to mark the memory aimed by a roll process, and (ii) to generate a notification $\mathsf{rl}\ k$ to freeze its continuation. Since the key k has been split into κ_1 and κ_2, in (4) the rule L.Branch is applied and then the notification $\mathsf{rl}\ k$ is split into the two sub-notification $\mathsf{rl}\ \kappa_1$ and $\mathsf{rl}\ \kappa_2$. Since in (5) the key κ_1 is contained into a memory, then the rule L.SPAN is applied and its effect are: freezing into the memory the process tagged by κ_1 and generating the notification $\mathsf{rl}\ k_4$ aiming to freeze the descendants of the memory. Since there is a notification on the key k_4 and a process in the environment tagged by k_4 we can apply the rule L.UP in (6) and freeze the process tagged by k_4. We apply the same rule on key κ_2 in (7) and we freeze the process tagged by κ_2. Since the continuation of the memory bearing the tag k_4 is frozen we can apply the rule L.STOP in (8) and we release the content of this memory. Now by using the structural law E.TAGPFR we can built back the process $k : (b\langle 0\rangle \mid \mathsf{roll}\ k)$ in (9). From (10) we apply again the rule L.STOP and we get back to the initial configuration. In general, a rollback of a step whose memory is tagged by k is performed by executing a top-down visit of its causal descendants, freezing them, followed by a bottom-up visit undoing the steps one at the time.

We now prove the correspondence between the DS semantics and the LL one. One can easily see that the two semantics are almost equal when transforming roll notifications of the form $\mathsf{rl}\ k, \kappa_1$ into $\mathsf{rl}\ \kappa_1$ and frozen configurations $\lfloor M \rfloor_S$ into $\lfloor M \rfloor$ if $S \neq \emptyset$ and M if $S = \emptyset$. The only remaining difference is that in the DS semantics roll notifications may proceed and add new keys to set S of frozen processes, while in the LL semantics they are blocked. However, these notifications will not have any effect on the rest of the system, and will finally disappear when facing a process which is outside a memory.

We can formalize this last intuition by showing that two LL processes differing only by some redundant rl notifications are bisimilar.

Lemma 4.7 *For each consistent LL configuration of the form $\mathbb{C}[\mathsf{rl}\ \kappa \mid M]$, we have $\mathbb{C}[\mathsf{rl}\ \kappa \mid M]\ {}_{LL}\overset{\cdot\ c}{\approx}_{LL}\ \mathbb{C}[M]$ if either M contains a notification $\mathsf{rl}\ \kappa$ or $\lfloor \kappa : P \rfloor \in M$ for some P.*

Proof: By coinduction, showing that related processes evolve to relate or identical processes. □

We can define a function $\gamma_d(\cdot)$ removing the redundant information.

Definition 4.13 *The function $\gamma_d(\cdot)$ from DS configurations to LL configu-*

rations is defined as follows:

$$\begin{array}{rcl@{\qquad}rcl}
\gamma_d(\nu u.\, M) & = & \nu u.\, \gamma_d(M) & \gamma_d(M \mid N) & = & \gamma_d(M) \mid \gamma_d(N) \\
\gamma_d(\kappa : P) & = & \kappa : P & \gamma_d(0) & = & 0 \\
\gamma_d([M; k]) & = & [M; k] & \gamma_d([M; k]^{\bullet}) & = & [M; k]^{\bullet} \\
\gamma_d(\lfloor \kappa : P \rfloor_{\emptyset}) & = & \kappa : P & \gamma_d(\lfloor \kappa : P \rfloor_{S}) & = & \lfloor \kappa : P \rfloor \ \textit{if } S \neq \emptyset \\
\gamma_d(\mathsf{rl}\ k, \kappa_1) & = & \mathsf{rl}\ \kappa_1 & & &
\end{array}$$

We can now prove the correspondence theorem.

Theorem 4.6 (Behavioral correspondence: DS vs LL) *For each (consistent) DS configuration M we have $M \;{}_{DS}\dot{\approx}^{c}_{LL}\; M$.*

Proof: We show that the relation:

$${}_{DS}\mathcal{R}_{LL} = \left\{ (\mathbb{C}[M], \mathbb{C}[\gamma_d(M)]) \,\middle|\, \begin{array}{c} M \text{ is a consistent DS configuration } \wedge \\ \mathbb{C}[M], \mathbb{C}[\gamma_d(M)] \text{ are consistent} \end{array} \right\}$$

is a weak barbed bisimulation with respect to DS and LL.

Barbs are trivially matched on both sides by definition of barb. Rules D.Com, D.Start, D.Stop, D.Branch are matched respectively by L.Com, L.Start, L.Stop, D.Branch and vice-versa. Rules D.Span and D.Up if applied on freeze-free configurations, that is with $S = \emptyset$, are matched respectively by rules L.Span and L.Up. Otherwise, since LL does not use labelling information, we can compose the result by transitivity using the Lemma 4.7 (because of redundant rl notifications) and then match those reductions with the identity, and vice-versa. □

We can finally concatenate all the results to obtain the correspondence between HL semantics and LL semantics.

Theorem 4.7 (Behavioral correspondence: HL vs LL) *For each (consistent) HL configuration M we have $M \;{}_{HL}\dot{\approx}^{c}_{LL}\; M$.*

Proof: We have the following results: $M \;{}_{HL}\dot{\approx}^{c}_{FR}\; M$ (Theorem 4.3), $M \;{}_{FR}\dot{\approx}^{c}_{RL}\; M$ (Theorem 4.4), $M \;{}_{RL}\dot{\approx}^{c}_{DS}\; M$ (Theorem 4.5), $M \;{}_{DS}\dot{\approx}^{c}_{LL}\; M$ (Theorem 4.6). The thesis follows by concatenating all the results and by using Lemma 4.3. □

This result can be easily formulated as full abstraction. In fact, the encoding j from HL configurations to LL configurations defined by the injection (identity) (HL configurations are a subset of LL configurations) is fully abstract.

Corollary 4.1 (Full abstraction) *Let j be the injection from HL (consistent) configurations to LL configurations and let M, N be two HL configurations. Then we have $j(M) \;{}_{LL}\dot{\approx}^{c}_{LL}\; j(N)$ iff $M \;{}_{HL}\dot{\approx}^{c}_{HL}\; N$.*

Proof: From Theorem 4.7 we have $M \;{}_{HL}\dot{\approx}^{c}_{LL}\; j(M)$ and $N \;{}_{HL}\dot{\approx}^{c}_{LL}\; j(N)$. The thesis follows by transitivity. □

The results above ensure that the loss of atomicity in rollback preserves the reachability of configurations yet does not make undesired configurations reachable.

Chapter 5

Encodings

5.1 Introduction

We show in this chapter that $\rho\pi$ can be encoded in a variant of HOπ, with bi-adic channels, join patterns [42, 43], sub-addressing, abstractions and applications which we call HOπ^+. This particular variant was chosen for convenience, because it simplifies our encoding. All HOπ^+ constructs are well understood in terms of expressive power with respect to HOπ (see [72,88] for abstractions in π-calculus).

The encoding presented in this chapter is slightly different from the one presented in [58], to obtain a finer result. Indeed [58] shows that a $\rho\pi$ configuration and its translation are equivalent by means of weak barbed bisimulation. Such result, allows us to encode reversibility in an already existing calculus, without using a specific ad-hoc primitive. Even if this result is quite surprising, because of the coarseness of weak barbed bisimulation (as emerged in Section 3.4), this equivalence is not a good relation on top of which build a faithfulness theorem for an encoding. Therefore, in this chapter we base our results on a stronger equivalence, able to distinguish forward reductions from backward ones.

The reminder of the chapter is organized as follows: first we introduce the syntax and the semantics of HOπ^+, then we introduce our encoding and show that a $\rho\pi$ consistent configuration M and its translation in HOπ^+ are equivalent by means of weak bf barbed bisimulation. To ease the reading of the chapter, some proofs are reported into Appendix A. Finally an encoding of roll-π into HOπ^+ is given and its correctness will be just conjectured.

5.2 HOπ^+

The syntax of HOπ^+ is given in Figure 5.1. Channels in HOπ^+ carry both an abstraction and a name (bi-adicity); a trigger can receive a message on a given channel, or on a given channel provided the received message carries some given name (sub-addressing). HOπ^+ has abstractions over names $(u)P$ and over process variables $(X)P$, and applications $(P\ V)$, where a value V can be a name or an abstraction. We take the set of names of HOπ^+ to be the set $\mathcal{I} \cup \{\star\}$ where $\mathcal{I}$ is the set of $\rho\pi$ identifiers. Thus both $\rho\pi$ names and $\rho\pi$ keys are names in HOπ^+. The set of (process) variables of HOπ^+ is taken to coincide with the set $\mathcal{V}$ of variables of $\rho\pi$.

$$\begin{aligned}
P, Q &::= \mathbf{0} \ \mid\ X \ \mid\ \nu u.\, P \ \mid\ (P \mid Q) \ \mid\ u\langle F, v\rangle \ \mid\ J \triangleright P \ \mid\ (F\ V)\\
F &::= (u)P \ \mid\ (X)P\\
V &::= u \ \mid\ F\\
J &::= u(X, v) \ \mid\ u(X, \backslash v) \ \mid\ J \mid J\\
u, v &\in \mathcal{I}
\end{aligned}$$

Figure 5.1: Syntax of HOπ^+.

The structural congruence for HOπ^+, noted $\equiv$, obeys the same rules as those of $\rho\pi$, except for the rules E.TagN and E.TagP, which are specific to $\rho\pi$. Evaluation contexts in HOπ^+ are given by the following grammar:

$$\mathbb{E} ::= \cdot \ \mid\ (P \mid \mathbb{E}) \ \mid\ \nu u.\, \mathbb{E}$$

The reduction relation for HOπ^+, also noted $\to$, is defined as the least evaluation closed relation (same definition as for $\rho\pi$, with HOπ^+ processes instead of configurations) that satisfies the rules in Figure 5.2, where ψ is either a name v, a process variable X or an escaped name $\backslash u$. We note $\Rightarrow$ the reflexive and transitive closure of $\to$. The function `match` in Figure 5.2 is the partial function which is defined in the cases given by the clauses below, and undefined otherwise:

$$\mathtt{match}(u, v) = \{^u/_v\} \qquad \mathtt{match}(u, \backslash u) = \{^u/_u\} \qquad \mathtt{match}(F, X) = \{^F/_X\}$$

let us note that an escaped name $\backslash u$ will match just with u. Rule Red is a generalization (in the sense of join patterns) of the usual communication rule for HOπ. If there are enough messages (left-hand side of the reduction in the conclusion) satisfying a certain input process, then the continuation of the input process is triggered with the necessary substitutions. Rule App mimics the β-reduction of the λ-calculus [10].

$$(\text{Red})\ \frac{\mathtt{match}(v_i, \psi_i) = \theta_i}{\prod_{i=1}^{n} u_i\langle F_i, v_i\rangle \mid (\prod_{i=1}^{n} u_i(X_i, \psi_i) \triangleright P) \to P\{^{F_1 \ldots F_n}/_{X_1 \ldots X_n}\}\theta_1 \ldots \theta_n}$$

$$(\text{App})\ ((\psi)F\ V) \to F\theta \qquad \mathtt{match}(V, \psi) = \theta$$

Figure 5.2: Reduction rules for HOπ^+.

Remark 5.1 *Even if the presented HOπ^+ allows the use of arbitrary join patterns, our encoding will use just binary join patterns.*

Conventions. In writing HOπ^+ terms, $u\langle v\rangle$ abbreviates $u\langle (X)\mathbf{0}, v\rangle$, $\overline{u}$ abbreviates $u\langle (X)\mathbf{0}, \star\rangle$ and $u\langle F\rangle$ abbreviates $u\langle F, \star\rangle$. Likewise, $a(u) \triangleright P$ abbreviates $a(X, u) \triangleright P$, where $X \notin \mathtt{fv}(P)$, $a \triangleright P$ abbreviates $a(X, \star) \triangleright P$, where $X \notin \mathtt{fv}(P)$, and $a(X) \triangleright P$ abbreviates $a(X, \star) \triangleright P$. We adopt the usual conventions for writing applications and abstractions: $(F\ V_1 \ldots V_n)$ stands for $(((F\ V_1) \ldots)\ V_n)$, and $(X_1 \ldots X_n)F$ stands for $(X_1) \ldots (X_n)F$. When there is no potential ambiguity, we often write $F\,V$ for $(F\ V)$. When defining HOπ^+ processes, we freely use recursive definitions for these can be encoded using e.g. the Turing fixed point combinator $\mathtt{\Theta}$ defined as $\mathtt{\Theta} = (\mathtt{A}\ \mathtt{A})$, where $\mathtt{A} = (X\ F)(F\ (X\ X\ F))$ (cf. [10] p.132).

In the rest of this chapter we note $\mathcal{P}_{\mathrm{HO}\pi^+}$ the set of HOπ^+ processes, $\mathcal{C}_{\rho\pi}$ the set of $\rho\pi$ configurations and $\mathcal{P}_{\rho\pi}$ the set of $\rho\pi$ processes.

5.3 $\rho\pi$ encoding

The encoding $(\!|\cdot|\!) : \mathcal{P}_{\rho\pi} \to \mathcal{P}_{\mathrm{HO}\pi^+}$ of processes of $\rho\pi$ in HOπ^+ is defined inductively in Figure 5.3. It extends to an encoding $(\!|\cdot|\!) : \mathcal{C}_{\rho\pi} \to \mathcal{P}_{\mathrm{HO}\pi^+}$ of configurations of $\rho\pi$ in HOπ^+ as given in Figure 5.4 (note that the encoding for $\mathbf{0}$ in Figure 5.4 is the encoding for the null configuration). The two main ideas behind the encoding are given now. First, a tagged process $l : P$ is interpreted as a process equipped with a special channel l (sometimes we will refer to it as *key channel*) on which to report that it has successfully rolled-back. This intuition leads to the encoding of a $\rho\pi$ process as an abstraction which takes this reporting channel l (its own key) as a parameter.

Second, each $\rho\pi$ process translation generates also the *process killer*, that is a process in charge of rolling it back. Hence we have three kinds of such processes: `KillM`, `KillT` and `KillP` representing respectively the killer of a message, of a trigger and of a parallel process.

Let us now describe the encoding of configurations given in Figure 5.4. A null configuration $\mathbf{0}$ is encoded as the null HOπ^+ process $\mathbf{0}$. The parallel and the restriction operator are mapped to the corresponding operators of HOπ^+. Depending on the kind of key, whether it is a simple name or a complex key, there are two ways to encode a tagged process $\kappa : P$. If the key is a simple name, then the translation is the application of the encoding of P to the name k, that is $(\!| P |\!)k$. If it is a complex key, for example of the form $\langle h_i, \tilde{h} \rangle \cdot k$, then the translation is the application of the translation of P to the name h_i, in parallel with the killer of the complex key. Since we want to generate at once a *tree* of killer processes able to revert an entire parallel composition made of n elements, with n being the size of $\tilde{h}$, we opt for the first element of the sequence to generate it. Said otherwise, a killer of a complex key is the null process if its h_i element is not the first one of the sequence $\tilde{h}$, otherwise it is a parallel composition of killer processes able to roll-back all the branches in which the simple key k has been split. Hence, the killer of the complex key, whose h_i element is the first one, is in charge of mimicking the behavior of the $\rho\pi$ structural rule E.TagP (see Figure 3.2 in Section 3.2.1) used (from right to left) to build back a tagged parallel composition from a parallel composition of (related) primitive processes. The encoding of a memory will be explained later.

Before commenting the encoding of $\rho\pi$ processes let us introduce the `Rew` process, and the idea behind it, crucial for the correct functioning of the encoding. Killer processes allow a process to roll-back but we have to give also the possibility to *undo* a rollback decision. This is due to the fact that at each moment we have to give the possibility to a process to continue forward or backward. If we think about a parallel composition of several processes, and one branch decides (spontaneously) to roll-back while the other branches do not, then the rolled-back process would be stuck unless we add the possibility to undo the rollback decision. This is actually the purpose of a process of the form (`Rew` l), whose behaviour is to read an abstraction carried in a message on the key channel l and then to re-instantiate the abstraction with the same key. Naturally this adds divergence to the encoding, but it is the price to pay to encode reversibility. Besides, since $\rho\pi$ is by itself a diverging calculus (a process can execute forever doing and undoing the same communication) this issue does not really matter.

Let us now comment on the encoding of $\rho\pi$ processes in Figure 5.3. As we already said, all the translations of $\rho\pi$ processes are abstractions over a channel, and this channel is the tag of the process (or part of it in the case of complex key). The zero process $\mathbf{0}$ is translated as a message on the abstracted channel along with a `Rew` process. Let us note that this translation is by itself diverging[1]. Consider the encoding of the $\rho\pi$ process $l : 0$

$$l\langle Nil\rangle \mid (\texttt{Rew}\; l) \to l\langle\texttt{Nil}\rangle \mid l(Z) \triangleright (Z\; l) \to (\texttt{Nil}\; l) \to l\langle\texttt{Nil}\rangle \mid (\texttt{Rew}\; l)$$

One may think to remove the `Rew` process from the translation of $\mathbf{0}$, and then to consider it as just a message on its abstracted channel. That is:

$$(\!|\mathbf{0}|\!) = (l)l\langle\texttt{Nil}\rangle$$

But in order to proving some invariants of our encoding, for symmetry with the translations of the other primitive processes (messages and triggers), we decided to add the `Rew` to the translation of the process $\mathbf{0}$.

The translation of a message $k : a\langle P\rangle$, is a process of the form[2]:

$$a\langle(\!|P|\!), k\rangle \mid (\texttt{KillM}\; a\; k)$$

consisting in a message on a carrying a pair (here we can see why we use bi-adic channels) in parallel with its killer process. The message carries the translation of the original message content P along with the abstracted channel. The need to add to the message content the abstracted channel k stems from the fact that the message will be rolled-back by just its own `KillM`. Indeed, the process $(\texttt{KillM}\; a\; k)$ just consumes a message on the channel a only if it carries the name k. This is why the `KillM` process is an abstraction over two channels, and explains why we use sub-addressing.

The translation of a parallel composition is quite straightforward: two new key channels are created and given to the translations of the two sub-processes, composing it, and then a `KillP` process will await on these two channels the rollback of the two sub processes, in order to notify its rollback by building back the entire parallel process into its key channel. This is why we use (binary) join patterns[3].

The translation of a trigger $l : a(X) \triangleright Q$, is a process of the form:

$$\nu\texttt{t}.\,\overline{\texttt{t}} \mid (a(X, h) \mid \texttt{t} \triangleright_f \nu k, c.\,(Y\; X\; c) \mid (c(Z) \triangleright (Z\; k)) \mid$$

[1] Actually all the translations of primitive processes are diverging.

[2] After several applications.

[3] The translation of triggers also uses join patterns.

$$(\mathtt{Mem}\, Y\ a\ X\ h\ k\ l)) \mid (\mathtt{KillT}\, Y\ \mathtt{t}\ l\ a)$$

with $Y = ((X\ c)c\langle (\!|Q|\!) \rangle)$. A special token $\overline{\mathtt{t}}$ is used as a lock in the translation of the trigger process. In this way either the trigger itself or its killer can acquire this lock and then execute by letting the other process blocked forever. Since all the messages on channel $a \in \mathcal{N}$ are translated in bi-adic messages, triggers are translated in order to read such messages. The continuation of a translated trigger mimics exactly the forward rule: it creates a new key channel k, it substitutes the variable X with the message content (that is an abstraction) in the trigger continuation. This substitution is mimicked by the application $(Y\ X\ c)$. For example suppose that after a communication we obtain the following process $Q\{^X/_P\}$ where Q is the body of the trigger. Then this substitution is mimicked in our encoding by the following application (and reduction):

$$((X\ c)c\langle (\!|Q|\!) \rangle)\ (\!|P|\!)\ c \to c\langle (\!|Q|\!)\{^{(\!|P|\!)}/_X\}\rangle$$

The trigger continuation, resulting from the substitution, is then applied to the new key channel (by the sub-process $c(Z) \triangleright Z\ k$) hence obtaining the process corresponding to the translation of $k : Q\{^P/_X\}$. Eventually a memory process Mem is created. The Mem process mimics exactly the backward rule of $\rho\pi$: it just awaits the rollback of its continuation (a message on the channel key channel k that the memory bears) and then it releases again the translations of the original $\rho\pi$ message and trigger who gave rise to the communication (and to the memory).

5.4 Correctness

This section is devoted to proving that the encoding is faithful, i.e. that it preserves the semantics of the original process. More precisely, we will prove the following theorem.

Theorem 5.1 (Faithfulness) *For any closed $\rho\pi$ process P, $\nu k.\ k : P \overset{\circ}{\approx} (\!|\nu k.\ k : P|\!)$.*

Before proving the theorem we will give a brief outline of our proof strategy.

Proof Outline: since the relation $\overset{\circ}{\approx}$ (see Definition 3.4) distinguishes three kinds of reduction: forward, backward and administrative, we first divide all the reductions induced by the encoding into these three kinds. Then we give a notion of *normal form* on HOπ^+ processes, whose intent is to

$$
\begin{array}{ll}
\llparenthesis \mathbf{0} \rrparenthesis = \texttt{Nil} & \llparenthesis X \rrparenthesis = X \\
\llparenthesis a\langle P\rangle \rrparenthesis = (l)(\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ l) & \llparenthesis \nu a.\, P \rrparenthesis = (l)\nu a.\, \llparenthesis P \rrparenthesis\, l \\
\llparenthesis P \mid Q \rrparenthesis = (l)(\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ l) \text{ if } P, Q \not\equiv 0 & \llparenthesis P \mid 0 \rrparenthesis = \llparenthesis P \rrparenthesis \\
\llparenthesis a(X) \triangleright P \rrparenthesis = (l)(\texttt{Trig}\ ((X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ l) & \llparenthesis 0 \mid P \rrparenthesis = \llparenthesis P \rrparenthesis
\end{array}
$$

$$
\begin{array}{l}
\texttt{Nil} = (l)(l\langle \texttt{Nil} \rangle \mid (\texttt{Rew}\ l)) \\
\texttt{Msg} = (a\ X\ l)a\langle X, l\rangle \mid (\texttt{KillM}\ a\ l) \\
\texttt{KillM} = (a\ l)(a(X, \backslash l) \triangleright l\langle (h)\texttt{Msg}\, a\, X\, h\rangle \mid \texttt{Rew}\, l) \\
\texttt{Par} = (X\ Y\ l)\nu h, k.\, X\ h \mid Y\ k \mid (\texttt{KillP}\ h\ k\ l) \\
\texttt{KillP} = (h\ k\ l)(h(W) \mid k(Z) \triangleright l\langle (l)\texttt{Par}\, W\, Z\, l\rangle \mid \texttt{Rew}\, l) \\
\texttt{Trig} = (Y\ a\ l)\nu \texttt{t}.\, \overline{\texttt{t}} \mid (a(X, h) \mid \texttt{t} \triangleright_f \nu k, c.\, (Y\ X\ c) \mid (c(Z) \triangleright (Z\ k)) \mid \\
\qquad\qquad (\texttt{Mem}\, Y\ a\ X\ h\ k\ l)) \mid (\texttt{KillT}\, Y\ \texttt{t}\ l\ a) \\
\texttt{KillT} = (Y\ \texttt{t}\ l\ a)(\texttt{t} \triangleright l\langle (h)\texttt{Trig}\, Y\, a\, h\rangle \mid \texttt{Rew}\, l) \\
\texttt{Mem} = (\ Y a\ X\ h\ k\ l)k(Z) \triangleright_b (\texttt{Msg}\ a\ X\ h) \mid (\texttt{Trig}\, Y\ a\ l) \\
\texttt{Rew} = (l)(l(Z) \triangleright Z\, l)
\end{array}
$$

Figure 5.3: Encoding $\rho\pi$ processes.

$$
\begin{array}{l}
\llparenthesis \mathbf{0} \rrparenthesis = \mathbf{0} \\
\llparenthesis M \mid N \rrparenthesis = \llparenthesis M \rrparenthesis \mid \llparenthesis N \rrparenthesis \\
\llparenthesis \nu u.\, M \rrparenthesis = \nu u.\, \llparenthesis M \rrparenthesis \\
\llparenthesis k : P \rrparenthesis = (\llparenthesis P \rrparenthesis\ k) \\
\llparenthesis \langle h_i, \tilde{h}\rangle \cdot k : P \rrparenthesis = (\llparenthesis P \rrparenthesis\ h_i) \mid \texttt{Kill}_{\langle h_i, \tilde{h}\rangle \cdot k} \\
\llparenthesis [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q; k] \rrparenthesis = (\texttt{Mem}\ ((X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)\ a\ \llparenthesis P \rrparenthesis\ \llparenthesis \kappa_1 \rrparenthesis\ k\ \llparenthesis \kappa_2 \rrparenthesis) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Kill}_{\kappa_1} \mid \texttt{Kill}_{\kappa_2} \\
\llparenthesis k \rrparenthesis = k \\
\llparenthesis \langle h_i, \tilde{h}\rangle \cdot k \rrparenthesis = h_i
\end{array}
$$

$$
\begin{array}{l}
\texttt{Kill}_{\langle h_1, \tilde{h}\rangle \cdot k} = \nu \tilde{l}.\, (\texttt{KillP}\ h_1\ l_1\ k) \mid \displaystyle\prod_{i=2}^{n-2} (\texttt{KillP}\ h_i\ l_i\ l_{i-1}) \mid (\texttt{KillP}\ h_{n-1}\ h_n\ l_{n-2}) \\
\texttt{Kill}_\kappa = \mathbf{0} \quad \text{otherwise}
\end{array}
$$

Figure 5.4: Encoding $\rho\pi$ configurations.

consider processes equivalent up to applications in active contexts. Then we characterize the garbage processes generated by the encoding with the function `addG`, because of the machinery added to simulate reversibility, and we characterize a new kind of congruence (noted $\equiv_{Ex}$) in order to have structurally congruent translations of structurally congruent $\rho\pi$ configurations. We then show how the reduction in normal form behaves with respect to the equivalence $\equiv_{Ex}$ and how the equivalence and administrative steps behave with respect to the reduction in normal form. We then show that a $\rho\pi$ reduction can be matched by the encoding, and that $\equiv_{Ex}$ is itself a weak backward and forward barbed bisimulation. Since encoding reductions add garbage, we prove that this garbage (characterized by `addG`) does not induce unwanted behaviors. That is, if P is a process derived from the encoding, then P and `addG`(P) are weakly bf barbed bisimilar. We then show that all the reductions of the encoding are matched by $\rho\pi$ and then compose all the obtained results to state our faithfulness theorem.

The first step for proving the theorem is to give to the LTS of the translation a backward and forward structure. In order to do that we have to partition the transitions of the translation into forward, backward and administrative. The basic idea is that administrative transitions can be used both as forward and as backward. Remember that in HOπ^+we have two kinds of reductions: applications and communications. Applications are always administrative. Communications are administrative, forward or backward according to the trigger that is involved in the communication. We extend HOπ^+triggers with labels to this end. This will not change the operational semantics of the calculus.

A labelled trigger is a trigger of the form $J \rhd_b P$ or $J \rhd_f P$. Triggers like the first one will be referred as backward ones while triggers like the second one will be referred as forward ones. We now define the two reduction relations $\twoheadrightarrow$ and $\rightsquigarrow$ on HOπ^+processes.

Definition 5.1 (Internal communications) *Let $\mapsto$ be the reduction relation involving communications due to non labelled triggers. Moreover, let $\mapsto^*$ the reflexive and transitive closure of $\mapsto$.*

Definition 5.2 (Internal trigger communication) *Let $\to_c$ the least evaluation closed relation defined on the following pair: $\{(c(Z) \rhd Z\ k \mid c\langle (\!| P |\!) \rangle\ ,\ (\!| P |\!) k)\}$.*

Definition 5.3 (Forward and backward HOπ^+relations) *Let $\to_f$ be a reduction involving a forward trigger and $\to_b$ a reduction involving a backward trigger. Moreover let $\hookrightarrow$ be the reduction relation on applications and on* *communications involving non labelled triggers. We define* $\twoheadrightarrow = \hookrightarrow^{*} \rightarrow_{f} \hookrightarrow^{*}$ *and* $\leadsto = \hookrightarrow^{*} \rightarrow_{b} \hookrightarrow^{*}$*.*

From the definitions above we have that $\rightarrow_{c} \subseteq \hookrightarrow$.

One cannot simply prove that given a (consistent) configuration M, if $M \rightarrow M'$ then $\llparenthesis M \rrparenthesis \Rightarrow \llparenthesis M' \rrparenthesis$. In fact this does not hold, since the translated processes produce some garbage (in terms of additional processes) due to the execution, and since structural congruent $\rho\pi$ processes do not always have structural congruent translations. Thus we need some auxiliary machinery.

First we characterize this kind of garbage and then we will give a new notion of structural congruence on processes generated by our encoding. We then define the function $\mathtt{addG}(P)$ that allows to add garbage to an HOπ^{+} process P.

Definition 5.4 *Let* P *be a HO*π^{+} *process such that* $P \equiv \nu\tilde{a}.\, P'$*. Then,* $\mathtt{addG}(P) \equiv \nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, Q)$*, where* Q *is a parallel composition (possibly empty) of processes of the form:*

$$\begin{array}{ll} \mathtt{Rew}\, l & \mathtt{KillM}\; a\; l \\ \nu c, \mathtt{t}.\, (\mathtt{KillT}\; ((X) c\langle \llparenthesis P \rrparenthesis \rangle)\; \mathtt{t}\; l\; a) & \nu \mathtt{t}.\, (a(X,k) | \mathtt{t} \triangleright S) \end{array}$$

The last two closures of the $\mathtt{addG}$ function characterizes the garbage processes produced by the consumption of the token $\overline{\mathtt{t}}$. Both processes are blocked, since the name $\mathtt{t}$ is restricted and the token $\overline{\mathtt{t}}$ has been consumed. For example:

$$\begin{array}{l} \llparenthesis k : a(X) \triangleright 0 \rrparenthesis = \llparenthesis a(X) \triangleright 0 \rrparenthesis k = ((h)\mathtt{Trig}\; Y\; a\; h)\; k \rightarrow (\mathtt{Trig}\; Y\; a\; k) \Rightarrow \\ \nu \mathtt{t}.\, \overline{\mathtt{t}} \mid (a(X, \backslash h) | \mathtt{t} \triangleright Q) \mid (\mathtt{t} \triangleright k\langle (h)\mathtt{Trig}\; Y\; a\; h \rangle \mid (\mathtt{Rew}\; k)) \rightarrow \\ \nu \mathtt{t}.\, (a(X, \backslash h) | \mathtt{t} \triangleright Q) \mid k\langle (h)\mathtt{Trig}\; Y\; a\; h \rangle \mid (\mathtt{Rew}\; k) \Rightarrow \\ \nu \mathtt{t}.\, (a(X, \backslash h) | \mathtt{t} \triangleright Q) \mid ((h)\mathtt{Trig}\; Y\; a\; h)\; k = \mathtt{addG}(\llparenthesis k : a(X) \triangleright 0 \rrparenthesis) \end{array}$$

with $Y = (X\ c) c\langle \llparenthesis 0 \rrparenthesis \rangle$ and $Q = \nu l, c.\, (Y\ X\ c) \mid (c(Z) \triangleright Z\ l) \mid (\mathtt{Mem}\; Y\; a\; X\; h\; l\; k)$. With the token trick, we avoid to use primitives such as *passivation* (see [64, 91]), in order to kill input processes, allowing us to encode reversibility in a simple but yet expressive calculus such as HOπ^{+}.

We now define a notion of normal form for processes, corresponding to processes where all the enabled applications have been executed. In particular, a process in normal form has no enabled applications.

Definition 5.5 (Normal form) *Let* $\mathtt{nf}(.)$ *be a function from* $\mathcal{P}_{HO\pi^+}$ *to* $\mathcal{P}_{HO\pi^+}$ *defined as follows:*

$$\begin{array}{ll}
\mathtt{nf}(\nu u.\, P) = \nu u.\, \mathtt{nf}(P) & \mathtt{nf}(P \mid Q) = \mathtt{nf}(P) \mid \mathtt{nf}(Q) \\
\mathtt{nf}(a\langle P\rangle) = a\langle P\rangle & \mathtt{nf}(a(X) \triangleright P) = a(X) \triangleright P \\
\mathtt{nf}((X)P\ Q) = \mathtt{nf}(P\{^{Q}/_{X}\}) & \mathtt{nf}((h)P\ l) = \mathtt{nf}(P\{^{l}/_{h}\}) \\
\mathtt{nf}(0) = 0 &
\end{array}$$

Lemma 5.1 *For each application* $(X)P\ Q$ *generated by the encoding either the process* P *is linear (variables are used once in* P*) or* P *is in normal form.*

Proof: The thesis does not hold for the encoding as presented in Figure 5.3, but it holds for the (completely equivalent) encoding obtained by replacing in the `Trig` macro ($\mathtt{KillT}\ Y\ \mathtt{t}\ l\ a$) by its normal form. One can note that in the new encoding the substitution of Y in `Trig` is the only non linear one, but its body is in normal form. □

We define a congruence on HOπ^+ processes to match the effect that $\rho\pi$ structural congruence has after the translation, in order to show that congruent $\rho\pi$ processes are translated into congruent HOπ^+ processes.

Definition 5.6 *Let* $\equiv_{Ax}$ *be the smallest congruence on HO*π^+ *processes satisfying the rules for structural congruence* $\equiv$ *plus the rules below.*

$$(\text{Ax.C})\ \mathtt{KillP}\ l\ h\ k \equiv_{Ax} \mathtt{KillP}\ h\ l\ k$$

$$(\text{Ax.P})\ l_1\langle(\!|P|\!)\rangle \mid l_2\langle(\!|Q|\!)\rangle \mid \mathtt{KillP}\ l_1\ l_2\ l \equiv_{Ax} l\langle(h)Par\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid$$

$$\mathtt{Rew}\ l \qquad \textit{with } P, Q \textit{ closed}$$

$$(\text{Ax.A})\ \nu l'.\,(\mathtt{KillP}\ l_1\ l_2\ l') \mid (\mathtt{KillP}\ l'\ l_3\ l) \equiv_{Ax} \nu l'.\,(\mathtt{KillP}\ l_1\ l'\ l) \mid (\mathtt{KillP}\ l_2\ l_3\ l')$$

$$(\text{Ax.Unfold})\ (\!|P|\!)l \equiv_{Ax} \nu\tilde{u}.\, l\langle(\!|Q|\!)\rangle \mid (\mathtt{Rew}\, l)\ \textit{with } P \equiv \nu\tilde{u}.\, Q$$

$$(\text{Ax.Adm})\ \nu c.\,(c\langle(\!|P|\!)\rangle \mid c(Z) \triangleright (Z\ k)) \equiv_{Ax} (\!|P|\!)k$$

Definition 5.7 *Let* $\equiv_{Ex}$ *be the smallest congruence including for each axiom* $L \equiv_{Ax} R$ *in* $\equiv_{Ax}$ *both* $L \equiv_{Ex} R$ *and* $\mathtt{nf}(L) \equiv_{Ex} \mathtt{nf}(R)$.

Axioms Ax.C and Ax.A are used to extend respectively the commutativity and associativity of the parallel operator also to the translation.

We now characterize the effect of the substitution on the structural congruence $\equiv_{Ex}$.

Lemma 5.2 *For any substitution $\sigma = \{{}^{l}/_{h}\}$ or $\sigma = \{{}^{(\!|P|\!)}/_{X}\}$, if $M \equiv_{Ex} N$ then $M\sigma \equiv_{Ex} N\sigma$.*

Proof: The thesis is trivial for name substitutions, thus we consider just higher-order ones. The proof is by induction on the derivation of $M \equiv_{Ex} N$, with a case analysis on the last applied axiom of $\equiv_{Ex}$. We will consider just a few significant cases.

$P \mid Q \equiv_{Ex} Q \mid P$**.** We have that $(P \mid Q)\sigma = P\sigma \mid Q\sigma \equiv_{Ex} Q\sigma \mid P\sigma$, as desired.

$(\nu u. P) \mid Q \equiv_{Ex} \nu u. (P \mid Q)$**.** We have that $((\nu u. P) \mid Q)\sigma = (\nu u. P)\sigma \mid Q\sigma$. Now if u is in the domain of σ we have that $(\nu u. P)\sigma \mid Q\sigma = (\nu u. P) \mid Q \equiv_{Ex} \nu u. (P \mid Q) = \nu u. (P \mid Q)\sigma$. Note that since u is not free in Q then the substitution does not affect it. If u is not in the domain of σ then banally we have that $(\nu u. P)\sigma \mid Q\sigma = (\nu u. P\sigma) \mid Q\sigma \equiv_{Ex} \nu u. (P\sigma \mid Q\sigma) = (\nu u. (P \mid Q))\sigma$, as desired. Note that by using the Barendregt's Variable Convention (see Remark 3.3 in Section 3.2) we assume that free variables are different from bound ones, so there is no need to check whether u is free in Q.

There is no need to consider the axioms in Definition 5.6 since they are all closed under process substitutions. □

We now characterize the effect of the normal form on the structural congruence $\equiv_{Ex}$.

Lemma 5.3 *If $M \equiv_{Ex} N$ then $\texttt{nf}(M) \equiv_{Ex} \texttt{nf}(N)$.*

Proof: Let us consider M and N generated by the encoding of Figure 5.3 where instead of having `Trig` processes, we substitute them with their normal form. Thus we can apply Lemma 5.1. Let us consider one application of an axiom. We have that $M = \mathbb{C}[L]$ and $N = \mathbb{C}[R]$ with $L \equiv_{Ex} R$ an axiom. By Lemma A.2 we have that $\texttt{nf}(\mathbb{C}[L]) = \mathbb{C}'[\texttt{nf}(L\sigma)]$ with $\texttt{nfc}(\mathbb{C}[\bullet], \emptyset) = \mathbb{C}'[\bullet], \sigma$, and the same with $\texttt{nf}(\mathbb{C}[R]) = \mathbb{C}'[\texttt{nf}(R\sigma)]$. By Lemma 5.2 we have that if $L \equiv_{Ex} R$ then $L\sigma \equiv_{Ex} R\sigma$. Also, $\texttt{nf}(L\sigma) \equiv_{Ex} \texttt{nf}(R\sigma)$ since $\equiv_{Ex}$ is closed under normal form. Finally, $\mathbb{C}'[\texttt{nf}(L\sigma)] \equiv_{Ex} \mathbb{C}'[\texttt{nf}(R\sigma)]$, as desired. □

Lemma 5.4 *If* $\texttt{nf}(T) \equiv_{Ex} \texttt{nf}(T')$ *then* $\texttt{nf}(\texttt{addG}(T)) \equiv_{Ex} \texttt{nf}(\texttt{addG}(T'))$.

Proof: See Appendix A.1. □

Lemma 5.5 *If* $\texttt{nf}(T_1) \equiv_{Ex} \texttt{nf}(T_2)$ *and* $\texttt{nf}(T_1) \to T_1'$ *then* $\texttt{nf}(T_2) \Rightarrow \texttt{nf}(T_2')$ *with* $\texttt{nf}(T_1') \equiv_{Ex} \texttt{nf}(T_2')$.

Proof: By case analysis on the used axiom $M \equiv_{Ex} N$ and on the structure of $\texttt{nf}(T_1)$. The proof can be found in Appendix A.1. □

We characterize now the effect of the encoding on structural congruent processes and configurations. We exploit to this end the structural congruence $\equiv_{Ex}$.

Lemma 5.6 *Let M, N be closed consistent configurations. Then $M \equiv N$ implies* $\texttt{nf}(⦇M⦈) \equiv_{Ex} \texttt{nf}(⦇N⦈)$.

Proof: By induction on the derivation of $M \equiv N$. The proof can be found in Appendix A.1. □

It is easy to see that names in $\mathcal{K}$ are always bound.

Lemma 5.7 *If* $⦇\nu k.\, k : P⦈ \Rightarrow P'$ *then* $\texttt{fn}(P') \cap \mathcal{K} = \emptyset$.

Proof: By induction on the number of reduction of $\Rightarrow$. Full proof can be found in Appendix A.2. □

In order to prove operational correspondence between $\rho\pi$ configurations and their translation we will start by showing that the the encoding is well-behaved w.r.t. substitutions.

Lemma 5.8 (Substitution) *For each $\rho\pi$ process P, Q:* $⦇P⦈\{^{⦇Q⦈}/_X\} = ⦇P\{^Q/_X\}⦈$.

Proof: By induction on the structure of P.

$P = 0$**:** we have $⦇0⦈\{^{⦇Q⦈}/_X\} = (l)(l\langle\texttt{Nil}\rangle \mid \texttt{Rew}\ l)\{^{⦇Q⦈}/_X\} = (l)(l\langle\texttt{Nil}\rangle \mid \texttt{Rew}\ l) = ⦇0⦈ = ⦇0\{^Q/_X\}⦈$ as desired.

$P = X$**:** we have $⦇X⦈\{^{⦇Q⦈}/_X\} = X\{^{⦇Q⦈}/_X\} = ⦇Q⦈ = ⦇X\{^Q/_X\}⦈$ as desired.

$P = Y$ **with** $X \neq Y$**:** we have $⦇Y⦈\{^{⦇Q⦈}/_X\} = Y\{^{⦇Q⦈}/_X\} = Y = ⦇Y⦈ = ⦇Y\{^Q/_X\}⦈$ as desired.

$P = a\langle P' \rangle$**:** we have that $\llparenthesis P \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Msg}\ a\ \llparenthesis P' \rrparenthesis\ l)\{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Msg}\ a\ \llparenthesis P' \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ l)$. By inductive hypothesis we have that $\llparenthesis P' \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\} = \llparenthesis P'\{^Q/_X\} \rrparenthesis$, and we have that $(l)(\texttt{Msg}\ a\ \llparenthesis P' \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ l) = (l)(\texttt{Msg}\ a\ \llparenthesis P'\{^Q/_X\} \rrparenthesis\ l) = \llparenthesis a\langle P'\{^Q/_X\}\rangle \rrparenthesis = \llparenthesis a\langle P' \rangle \{^Q/_X\} \rrparenthesis$ as desired.

$P = a(Y) \triangleright P'$**:** we have that $\llparenthesis P \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Trig}\ ((Y\ c)c\langle \llparenthesis P' \rrparenthesis \rangle)\ a\ l)\{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Trig}\ ((Y\ c)c\langle \llparenthesis P' \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\rangle)\ a\ l)$, and by using the inductive hypothesis we have that $(l)(\texttt{Trig}\ ((Y\ c)c\langle \llparenthesis P' \rrparenthesis \rangle)\ a\ l)\{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Trig}\ ((Y\ c)c\langle \llparenthesis P' \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\rangle)\ a\ l) = (l)(\texttt{Trig}\ ((Y\ c)c\langle \llparenthesis P'\{^Q/_X\} \rrparenthesis \rangle)\ a\ l) = \llparenthesis a(Y) \triangleright P'\{^Q/_X\} \rrparenthesis = \llparenthesis a(Y) \triangleright P'\{^Q/_X\} \rrparenthesis$ as desired.

$P = P_1 \mid P_2$**:** if P_1 or P_2 are equivalent to 0 we can directly conclude by applying the inductive hypothesis. In the other case we have that $\llparenthesis P \rrparenthesis = (l)(\texttt{Par}\ \llparenthesis P_1 \rrparenthesis\ \llparenthesis P_2 \rrparenthesis\ l)\{^{\llparenthesis Q \rrparenthesis}/_X\} = (l)(\texttt{Par}\ \llparenthesis P_1 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ \llparenthesis P_2 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ l)$ and by applying the inductive hypothesis, $\llparenthesis P_1 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\} = \llparenthesis P_1\{^Q/_X\} \rrparenthesis$ and $\llparenthesis P_2 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\} = \llparenthesis P_2\{^Q/_X\} \rrparenthesis$ we have $(l)(\texttt{Par}\ \llparenthesis P_1 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ \llparenthesis P_2 \rrparenthesis \{^{\llparenthesis Q \rrparenthesis}/_X\}\ l) = (l)(\texttt{Par}\ \llparenthesis P_1\{^Q/_X\} \rrparenthesis\ \llparenthesis P_2\{^Q/_X\} \rrparenthesis\ l) = \llparenthesis (P_1 \mid P_2)\{^Q/_X\} \rrparenthesis$ as desired.

□

We now show that communications can always be postponed with respect to the applications, this will help us to state that if a generic HOπ^+ process derived by our encoding can perform a communication then also its normal form can do the same communication. We now define a reduction relation on applications of particular forms, generated by our encoding, namely applications of a channel to an abstraction over a channel and application of a translation, generated by our encoding, to an abstraction over a process.

Definition 5.8 *Let $\rightharpoondown$ the least evaluation closed relation defined on the following pairs: $\{\langle (h)P\ v, P\{^v/_h\}\rangle \mid P \in \mathcal{P}\} \cup \{(X)P\ \llparenthesis Q \rrparenthesis\ , P\{^{\llparenthesis Q \rrparenthesis}/_X\} \mid P, \llparenthesis Q \rrparenthesis \in \mathcal{P}\}$. Moreover let $\rightharpoondown^*$ be the reflexive and transitive closure of $\rightharpoondown$.*

From now on we will denote with $\rightarrow_?$ the reflexive closure of $\rightarrow$, where $\rightarrow$ is the HOπ^+ reduction relation. We will do similarly for other relations.

Lemma 5.9 *If $P \rightarrow_? P'$ and $P \rightharpoondown_? P''$ then $P'' \rightarrow_? Q$ and $P' \rightharpoondown_? Q$.*

We need an auxiliary lemma before the proof.

Lemma 5.10 *If $\mathbb{E}[(\psi)F\ V] \rightarrow T$ then either:*

- $\mathbb{E}[0] \rightarrow \mathbb{E}'[0]$ *with* $T = \mathbb{E}'[(\psi)F\ V]$ *or*

- $\mathbb{E}[(\psi)F\ V] \rightarrow \mathbb{E}[R]$ *with* $(\psi)F\ V \rightarrow R$

Proof: We have that the sole possible reductions that can be performed by a HOπ^{+}process are applications and communications. If $\mathbb{E}[(\psi)F\ V] \rightarrow T$ via a communication this implies that the process in the hole is not part of the communication nor it gives raise to a communication. This is due to the fact that the context is active, so the process in the hole cannot be the content of a message or the continuation of an input process. So we have that $\mathbb{E}[(\psi)F\ V] \rightarrow \mathbb{E}'[(\psi)F\ V]$, but this also implies that $\mathbb{E}[0] \rightarrow \mathbb{E}'[0]$. The same reasoning can be applied if the context $\mathbb{E}$ performs an application step different from the one contained in the hole. If we apply the application inside the hole we will banally have that $\mathbb{E}[(\psi)F\ V] \rightarrow \mathbb{E}[R]$ with $(\psi)F\ V \rightarrow R$ and we are done.

□

Proof of lemma 5.9: The cases in which $\rightarrow$ or $\rightharpoondown$ are not applied are clear and the property trivially holds. The case in which $\rightarrow$ and $\rightharpoondown$ are the same (they are applied on the same application) is trivially verified. Now let us consider cases in which the two reductions happens and then we proceed on case analysis on $\rightarrow$. If the process P can perform an application this means that we can write P as an execution context containing an application, so $P = \mathbb{E}[(\psi)F\ V]$. Now by Lemma 5.10 we know that if $\mathbb{E}[(\psi)F\ V] \rightarrow T$ than either $\mathbb{E}[0] \rightarrow \mathbb{E}'[0]$ with $T = \mathbb{E}'[(\psi)F\ V]$ or $\mathbb{E}[(\psi)F\ V] \rightarrow \mathbb{E}[R]$ with $(\psi)F\ V \rightarrow R$. So if $P \rightarrow P'$ via a communication (or an application different from the one in the hole) we have that $P' = \mathbb{E}'[(\psi)F\ V]$ but also $P \rightharpoondown P''$ with $P'' = \mathbb{E}[R]$. Now we can easily see that P' may perform the hole application and then $P' \rightharpoondown \mathbb{E}'[R]$ and P'' can perform the reduction $P'' \rightarrow \mathbb{E}'[R]$ (using as hypothesis $\mathbb{E}[0] \rightarrow \mathbb{E}'[0]$), as desired. □

Lemma 5.11 *If* $P \Rightarrow P'$ *and* $P \rightharpoondown^{*} P''$ *then* $P' \rightharpoondown^{*} Q$ *and* $P'' \Rightarrow Q$.

Proof: By simple induction on the length of $\Rightarrow$ and $\rightharpoondown^{*}$ and by applying lemma 5.9 for the base case. □

Two concurrent applications can always be swapped, according to the following Lemma.

Lemma 5.12 *If* $N \rightharpoondown N_1$ *and* $N \rightharpoondown N_2$ *then* $N_2 \rightharpoondown N_3$ *and* $N_1 \rightharpoondown N_3$

Proof: If N can perform two independent applications this means that we can write N as a binary execution context of the form $\mathbb{D}[(\psi_1)F_1\ V_1\ ,\ (\psi_2)F_2\ V_2]$.

$$\begin{tikzcd}
M & \equiv & M' & & & & \\
& 5.6 & & & & & \\
\mathtt{nf}(\llparenthesis M \rrparenthesis) \arrow[dd, twoheadrightarrow] & \equiv_{Ex} & \mathtt{nf}(\llparenthesis M' \rrparenthesis) \arrow[dd, twoheadrightarrow, "ind"] & & N' & \equiv & N \\
& 5.5 & & & & 5.6 & \\
Q \arrow[dd, "*"] & & P \arrow[dd, "*"] & & \mathtt{nf}(\llparenthesis N' \rrparenthesis) & \equiv_{Ex} & \mathtt{nf}(\llparenthesis N \rrparenthesis) \\
& & & & \equiv_{Ex} & 5.4 & \equiv_{Ex} \\
\mathtt{nf}(Q) & \equiv_{Ex} & \mathtt{nf}(P) & \equiv_{Ex} & \mathtt{nf}(\mathtt{addG}(\llparenthesis N' \rrparenthesis)) & \equiv_{Ex} & \mathtt{nf}(\mathtt{addG}(\llparenthesis N \rrparenthesis))
\end{tikzcd}$$

Figure 5.5: Confluence of encoding with respect to $\equiv_{Ex}$ (numbers refer to Lemmas).

Now let us suppose that N_1 is the process in which the left application has been done and N_2 the process in which the right application has been performed. Hence $N \rightharpoondown \mathbb{D}[Q_1\ ,\ (\psi_2)F_2\ V_2] = N_1$ with $(\psi_1)F_1\ V_1 \rightharpoondown Q_1$ and $N \rightharpoondown \mathbb{D}[(\psi_1)F_1\ V_1\ ,\ Q_2] = N_2$ with $(\psi_2)F_2\ V_2 \rightharpoondown Q_2$. Now we see that N_1 can still perform the right application, and N_2 the left one. So $N_1 \rightharpoondown \mathbb{D}[Q_1\ ,\ Q_2]$ and $N_2 \rightharpoondown \mathbb{D}[Q_1\ ,\ Q_2]$, as desired. The other cases are similar. □

We now prove an important property of our encoding, showing the importance of the key channels. Essentially, a translation of a $\rho\pi$ process P can always roll-back, and this is represented by a message on the process key channel. The roll-back is not *perfect* in the sense that the content of the message is not equal to the translation of the original process P. This is due to the translation of the name creation process, since once a name is created there is no way to reverse it. However the property that we can state is that the process composed by the extruded names and the content of the message is structurally equivalent to the original process P. Formally we have:

Lemma 5.13 *For each closed $\rho\pi$ process P, $\llparenthesis P \rrparenthesis k \hookrightarrow^* \nu\tilde{u}.\, k\langle \llparenthesis Q \rrparenthesis \rangle \mid \mathtt{Rew}\ k \mid S$ with $k \notin \tilde{u}$, $S = \prod R_i$, $R_i = \mathtt{Rew}\ k_i$ or $R_i = \nu\mathtt{t}.\,(a(X,h)|\mathtt{t}\triangleright R)$ and $P \equiv \nu\tilde{u}.\, Q$*

Proof: By induction on the structure of P.

$P = 0$ **:** we have that $\llparenthesis 0 \rrparenthesis k = (l)(l\langle \mathtt{Nil} \rangle \mid \mathtt{Rew}\ l)k \rightharpoondown k\langle \mathtt{Nil} \rangle \mid \mathtt{Rew}\ k = k\langle \llparenthesis 0 \rrparenthesis \rangle \mid \mathtt{Rew}\ k$, as desired.

$P = a\langle Q\rangle$ **:** we have that

$$\llparenthesis P\rrparenthesis k = (h)(\texttt{Msg}\ a\ \llparenthesis Q\rrparenthesis\ h)k \rightharpoondown \texttt{Msg}\ a\ \llparenthesis Q\rrparenthesis\ k \rightharpoondown a\langle \llparenthesis Q\rrparenthesis, k\rangle \mid \texttt{KillM}\ a\ k \rightharpoondown$$
$$a\langle \llparenthesis Q\rrparenthesis, k\rangle \mid (a(X, \backslash k) \triangleright k\langle (h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ k) \hookrightarrow$$
$$k\langle (h)(\texttt{Msg}\ a\ \llparenthesis Q\rrparenthesis\ h)\rangle \mid \texttt{Rew}\ k = k\langle \llparenthesis P\rrparenthesis\rangle \mid \texttt{Rew}\ k = k\langle \llparenthesis P\rrparenthesis\rangle \mid S$$

as desired.

$P = a(X) \triangleright P'$ **:** be $Y = (X\ c)c\langle \llparenthesis P'\rrparenthesis\rangle$, we have that

$$\llparenthesis P\rrparenthesis k = ((l)(\texttt{Trig}\ Y\ a\ l))k \rightharpoondown \texttt{Trig}\ Y\ a\ k \rightharpoondown$$
$$\nu\texttt{t}.\,(\overline{\texttt{t}} \mid a(X,h)|\texttt{t} \triangleright R \mid (\texttt{KillT}\ Y\ \texttt{t}\ k\ a)) \rightharpoondown$$
$$\nu\texttt{t}.\,(\overline{\texttt{t}} \mid a(X,h)|\texttt{t} \triangleright R \mid \texttt{t} \triangleright k\langle (h)\texttt{Trig}\ Y\ a\ h\ c\rangle \mid \texttt{Rew}\ k) \hookrightarrow$$
$$\nu\texttt{t}.\, a(X,h)|\texttt{t} \triangleright R \mid k\langle (h)\texttt{Trig}\ Y\ a\ h\rangle \mid \texttt{Rew}\ k \equiv$$
$$(\nu\texttt{t}.\, a(X,h)|\texttt{t} \triangleright R) \mid k\langle (h)\texttt{Trig}\ Y\ a\ h\rangle \mid \texttt{Rew}\ k = k\langle (h)\texttt{Trig}\ Y\ a\ h\rangle \mid S$$

as desired.

$P = X$ **:** this case is never applied since we are considering closed processes.

$P = \nu a.\, P'$ **:** we have that $\llparenthesis \nu a.\, P'\rrparenthesis k = ((h)\nu a.\, \llparenthesis P'\rrparenthesis h)k \rightharpoondown \nu a.\, \llparenthesis P'\rrparenthesis k$. Now by inductive hypothesis we know that $\llparenthesis P'\rrparenthesis k \hookrightarrow^* \nu\tilde{u}.\, k\langle \llparenthesis Q\rrparenthesis\rangle \mid S$ with $P' \equiv \nu\tilde{u}.\, Q$ and since restriction is an execution context we have $\nu a.\, \llparenthesis P'\rrparenthesis k \hookrightarrow^* \nu a.\, \nu\tilde{u}.\, k\langle \llparenthesis Q\rrparenthesis\rangle \mid S$ with $\nu a.\, P' \equiv \nu a.\, \nu\tilde{u}.\, Q$, as desired.

$P = P_1 \mid P_2$ **:** we have that

$$\llparenthesis P\rrparenthesis k = ((l)(\texttt{Par}\ \llparenthesis P_1\rrparenthesis\ \llparenthesis P_2\rrparenthesis\ l)k \rightharpoondown \texttt{Par}\ \llparenthesis P_1\rrparenthesis\ \llparenthesis P_2\rrparenthesis\ k \rightharpoondown$$
$$\nu h, l.\, \llparenthesis P_1\rrparenthesis h \mid \llparenthesis P_2\rrparenthesis l \mid \texttt{KillP}\ l\ h\ k \rightharpoondown$$
$$\nu h, l.\, \llparenthesis P_1\rrparenthesis h \mid \llparenthesis P_2\rrparenthesis l \mid (h(W)|l(Z) \triangleright k\langle (h)\texttt{Par}\ W\ Z\ h\rangle \mid \texttt{Rew}\ k).$$

By inductive hypothesis we have that $\llparenthesis P_1\rrparenthesis h \hookrightarrow^* \nu\tilde{u}.\, h\langle \llparenthesis P_1'\rrparenthesis\rangle \mid S_1$ with $P_1 \equiv \nu\tilde{u}.\, P_1'$ and $\llparenthesis P_2\rrparenthesis l \hookrightarrow^* \nu\tilde{v}.\, l\langle \llparenthesis P_2'\rrparenthesis\rangle \mid S_2$ with $P_2 \equiv \nu\tilde{v}.\, P_2'$. Hence we have

$$\nu h, l.\, \llparenthesis P_1\rrparenthesis h \mid \llparenthesis P_2\rrparenthesis l \mid h(W)|l(Z) \triangleright (k\langle (h)\texttt{Par}\ W\ Z\ h\rangle \mid \texttt{Rew}\ k) \hookrightarrow^*$$
$$\nu h, l, \tilde{u}, \tilde{v}.\, h\langle \llparenthesis P_1'\rrparenthesis\rangle \mid S_1 \mid l\langle \llparenthesis P_2'\rrparenthesis\rangle \mid S_2 \mid h(W)|l(Z) \triangleright k\langle (h)\texttt{Par}\ W\ Z\ h\rangle \mid \texttt{Rew}\ k \hookrightarrow$$
$$\nu h, l, \tilde{u}, \tilde{v}.\, S_1 \mid S_2 \mid k\langle (h)\texttt{Par}\ P_1'\ P_2'\ h\rangle \mid \texttt{Rew}\ k \equiv$$
$$\nu h, l, \tilde{u}, \tilde{v}.\, k\langle (h)\texttt{Par}\ \llparenthesis P_1'\rrparenthesis\ \llparenthesis P_2'\rrparenthesis\ h\rangle \mid S$$

with $S = \texttt{Rew}\ k \mid S_1 \mid S_2$ and by garbage collecting names h, l we have

$P \equiv \nu\tilde{u}, \tilde{v}.\, (P_1' \mid P_2')$ as desired.

□

The next theorem shows that the encoding of a process can mimic its reductions.

Theorem 5.2 *For each consistent configuration M, if $M \to N$ then* $\mathtt{nf}(\llparenthesis M \rrparenthesis) \to P$ *with* $\mathtt{nf}(P) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(\llparenthesis N \rrparenthesis))$.

Proof: By induction on the derivation $M \to N$ with a case analysis on the last rule applied.

R.Fw: we have that

$$\begin{aligned} M = \kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q \twoheadrightarrow \\ \nu k.\, (Q\{^P/_X\} \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q\ ;\ k]) = N \end{aligned}$$

Moreover we have that $\llparenthesis M \rrparenthesis = \llparenthesis \kappa_1 : a\langle P\rangle \rrparenthesis \mid \llparenthesis \kappa_2 : a(X) \triangleright Q \rrparenthesis$. Now depending whether κ_1, κ_2 are complex or not there are different cases to be treated. Let us suppose that $\kappa_1 = k_1$ and $\kappa_2 = k_2$ and $Y = ((X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)$, so

$$\begin{aligned} &\mathtt{nf}(\llparenthesis M \rrparenthesis) = \mathtt{nf}((l)(\mathtt{Msg}\, a\ \llparenthesis P \rrparenthesis\ l)k_1) \mid \mathtt{nf}((l)(\nu \mathtt{t}.\, \mathtt{Trig}\, Y\ a\ l)) = \\ &\nu \mathtt{t}.\, a\langle \llparenthesis P \rrparenthesis, k_1\rangle \mid \mathtt{nf}(\mathtt{KillM}\, a\ k_1) \mid \overline{\mathtt{t}} \mid \\ &\quad (\mathtt{t}|a(X,h) \triangleright_f \nu k, c.\, (Y\ X\ c) \mid c(Z) \triangleright Z\ k \mid (\mathtt{Mem}\, Y\ a\ X\ h\ k\ k_2)) \mid \\ &\quad \mathtt{nf}(\mathtt{KillT}\, Y\ a\ k_1) \to_f \\ &\nu c, k, \mathtt{t}.\, \mathtt{nf}(\mathtt{KillM}\, a\ k_1) \mid (Y\ \llparenthesis P \rrparenthesis\ c) \mid (c(Z) \triangleright Z\ k) \mid (\mathtt{Mem}\, Y\ a\ X\ k_1\ k\ k_2) \mid \\ &\qquad\quad \mathtt{nf}(\mathtt{KillT}\, Y\ a\ k_2) \rightharpoondown \\ &\nu c, k, \mathtt{t}.\, \mathtt{nf}(\mathtt{KillM}\, a\ k_1) \mid c\langle \llparenthesis Q \rrparenthesis \{^{\llparenthesis P \rrparenthesis}/_X\}\rangle \mid (c(Z) \triangleright Z\ k) \mid \\ &\qquad\quad (\mathtt{Mem}\, Y\ a\ X\ k_1\ k\ k_2) \mid \mathtt{nf}(\mathtt{KillT}\, Y\ a\ k_2) \hookrightarrow \\ &\nu c, k, \mathtt{t}.\, (\mathtt{KillM}\, a\ k_1) \mid (\llparenthesis Q \rrparenthesis \{^{\llparenthesis P \rrparenthesis}/_X\}\ k) \mid (\mathtt{Mem}\, Y\ a\ X\ k_1\ k\ k_2) \mid \\ &\qquad\quad \mathtt{nf}(\mathtt{KillT}\, Y\ a\ k_2) = M' \end{aligned}$$

Now by using Lemma 5.8 we have that $\llparenthesis Q \rrparenthesis \{^{\llparenthesis P \rrparenthesis}/_X\} = \llparenthesis Q\{^P/_X\} \rrparenthesis$ and then

$$\begin{aligned} M' = \nu c, k, \mathtt{t}.\, \mathtt{nf}(\mathtt{KillM}\, a\ k_1) \mid (\llparenthesis Q\{^P/_X\} \rrparenthesis\ k) \mid \\ (\mathtt{Mem}\, Y\ a\ X\ k_1\ k\ k_2) \mid \mathtt{nf}(\mathtt{KillT}\, Y\ a\ k_2) \end{aligned}$$

We can conclude by noticing that

$$\texttt{nf}(M') = \texttt{nf}(\texttt{KillM}\ a\ k_1) \mid \texttt{nf}((\!|N|\!)) \mid \nu c.\, \texttt{nf}(\texttt{KillT}\ Y\ a\ k_2) = \texttt{nf}(\texttt{addG}((\!|N|\!)))$$

as desired.

Let us consider the case in which $\kappa_1 = \langle h_i, \tilde{h}\rangle \cdot k$, so we have that

$$(\!|M|\!) = (l)(\texttt{Msg}\ a\ (\!|P|\!)\ l)h_i \mid \texttt{Kill}_{\langle h_i,\tilde{h}\rangle\cdot k} \mid (l)(\nu c, \texttt{t}.\, \texttt{Trig}\ Y\ a\ l\ c)k_2$$

Now using the same reductions of the above case we have that

$$\begin{aligned}&\texttt{nf}((\!|M|\!)) \twoheadrightarrow\\ &\nu c, k, \texttt{t}.\, \texttt{nf}(\texttt{KillM}\ a\ k_1) \mid \texttt{nf}(\texttt{Kill}_{\langle h_i,\tilde{h}\rangle\cdot k}) \mid (\!|Q|\!)\{^{(\!|P|\!)}/_X\}\ k \mid\\ &\qquad (\texttt{Mem}\ Y\ a\ X\ h_i\ k\ k_2) \mid \texttt{nf}(\texttt{KillT}\ Y\ a\ k_2) = M'\end{aligned}$$

and by using Lemma 5.8 we have that

$$\begin{aligned}M' = \nu c, k, \texttt{t}.\, \texttt{nf}(\texttt{KillM}\ a\ h_i) \mid \texttt{nf}(\texttt{Kill}_{\langle h_i,\tilde{h}\rangle\cdot k}) \mid ((\!|Q\{^{(\!|P|\!)}/_X\}|\!)\ k) \mid\\ (\texttt{Mem}\ Y\ a\ X\ h_i\ k\ k_2) \mid \texttt{nf}(\texttt{KillT}\ Y\ a\ k_2)\end{aligned}$$

We can conclude by noticing that $\texttt{nf}(M') = \texttt{nf}(\texttt{addG}((\!|N|\!)))$ as desired. The other two cases with complex keys are similar.

R.Bw we have that $M = k : R \mid [\kappa_1 : a\langle P\rangle \mid \kappa_2 : a(X) \triangleright Q\ ;\ k] \rightsquigarrow N$. Let suppose that $Y = ((X\ c)c\langle(\!|Q|\!)\rangle)$, $\kappa_1 = \langle h', \tilde{h}'\rangle \cdot k_1$ and $\kappa_2 = \langle h'', \tilde{h}''\rangle \cdot k_2$. Then by definition of $\texttt{nf}(\_)$ we have

$$\begin{aligned}&\texttt{nf}((\!|M|\!)) =\\ &\texttt{nf}((\!|R|\!)k) \mid \texttt{nf}(\texttt{Mem}\ Y\ a\ (\!|P|\!)\ (\!|\kappa_1|\!)\ k\ (\!|\kappa_2|\!)) \mid \texttt{nf}(\texttt{Kill}_{\kappa_1}) \mid \texttt{nf}(\texttt{Kill}_{\kappa_2}) =\\ &\texttt{nf}((\!|R|\!))k \mid (k(Z) \triangleright (\texttt{Msg}\ a\ (\!|P|\!)\ h') \mid (\texttt{Trig}\ Y\ a\ h'')) \mid \texttt{Kill}_{\kappa_1} \mid \texttt{Kill}_{\kappa_2}\end{aligned}$$

By Lemma 5.13 [4] we know that $\texttt{nf}((\!|R|\!)k) \hookrightarrow^* \nu\tilde{u}.\, k\langle(\!|R'|\!)\rangle \mid \texttt{nf}(S) \mid \texttt{Rew}\ k$ with S being a parallel composition of certain garbage and

[4] Lemma 5.13 refers to processes not in normal form, but its extension to processes in normal form is trivial.

$R \equiv \nu\tilde{u}.\, R'$. Hence

$$\begin{aligned}&\mathtt{nf}(\llparenthesis R \rrparenthesis k) \mid (k(Z) \triangleright (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h') \mid (\mathtt{Trig}\ Y\ a\ h'')) \mid\\ &\mathtt{nf}(\mathtt{Kill}_{\kappa_1}) \mid \mathtt{nf}(\mathtt{Kill}_{\kappa_2}) \hookrightarrow^*\\ &\nu\tilde{u}.\, k\langle \llparenthesis R' \rrparenthesis \rangle \mid \mathtt{nf}(S) \mid \mathtt{Rew}\ k \mid (k(Z) \triangleright (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h') \mid (\mathtt{Trig}\ Y\ a\ h'')) \mid\\ &\quad \mathtt{nf}(\mathtt{Kill}_{\kappa_1}) \mid \mathtt{nf}(\mathtt{Kill}_{\kappa_2}) \hookrightarrow\\ &\nu\tilde{u}.\, (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h') \mid (\mathtt{Trig}\ Y\ a\ h'') \mid \mathtt{nf}(\mathtt{Kill}_{\kappa_1}) \mid \mathtt{nf}(\mathtt{Kill}_{\kappa_2}) \mid\\ &\quad \mathtt{nf}(S) \mid \mathtt{Rew}\ k = M'\end{aligned}$$

We have that $\mathtt{nf}(M') \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(\llparenthesis \kappa_1 : a\langle P \rangle \mid \kappa_2 : a(X) \triangleright Q \rrparenthesis))$, as desired.

Equiv: we have that $M \twoheadrightarrow N$ with hypothesis $M \equiv M'$ $M' \twoheadrightarrow N'$ and $N' \equiv N$. Figure 5.5 (numbers refers to the used lemmas and *ind* means that inductive hypothesis are applied) shows the proof schema we use to prove this case. By inductive hypothesis we have that $\llparenthesis M' \rrparenthesis \twoheadrightarrow P$ with $\mathtt{nf}(P) \equiv_{Ex} \mathtt{addG}(\mathtt{nf}(\llparenthesis N' \rrparenthesis))$, and by hypothesis we have that $M \equiv M'$ and $N' \equiv N$. By Lemma 5.6, we have that if $M \equiv M'$ then $\mathtt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \mathtt{nf}(\llparenthesis M' \rrparenthesis)$, and by Lemma 5.5 we have that if $\mathtt{nf}(\llparenthesis M' \rrparenthesis) \Rightarrow \mathtt{nf}(P)$ then $\mathtt{nf}(\llparenthesis M \rrparenthesis) \Rightarrow \mathtt{nf}(Q)$ with $\mathtt{nf}(Q) \equiv_{Ex} \mathtt{nf}(P)$. By inductive hypothesis we have that $\mathtt{nf}(P) \equiv_{Ex} \mathtt{addG}(\mathtt{nf}(\llparenthesis N' \rrparenthesis))$ but since by hypothesis we had $N' \equiv N$ by Lemma 5.6 we have that $\mathtt{nf}(\llparenthesis N' \rrparenthesis) \equiv_{Ex} \mathtt{nf}(\llparenthesis N \rrparenthesis)$ and by Lemma 5.4 we have that $\mathtt{addG}(\mathtt{nf}(\llparenthesis N' \rrparenthesis)) \equiv_{Ex} \mathtt{addG}(\mathtt{nf}(\llparenthesis N \rrparenthesis))$. So we can conclude by saying that $\llparenthesis M \rrparenthesis \twoheadrightarrow Q$ and that $\mathtt{nf}(Q) \equiv_{Ex} \mathtt{addG}(\mathtt{nf}(\llparenthesis N \rrparenthesis))$, as desired. The case in which a backward reduction is applied is similar.

Ctx: by simply induction on the structure of the context, noting that the translation of active contexts is isomorphic.

□

The following lemma is the equivalent of the Loop Lemma for the encoding. Essentially, it states that any process P derived from the translation of a consistent configuration M, can reverse all the administrative steps that it has performed. Hence if $P \hookrightarrow^* Q$ then from Q there exist an execution that allows us to get back into a process that is *somehow* equivalent to P. Naturally we have to take into account all the garbage processes (killers and `Rew` processes) that administrative steps may generate. This will help us to reason up-to administrative steps, and to consider P and Q as equivalent.

Lemma 5.14 *For any consistent configuration M, $(\!|M|\!) \Rightarrow P$, if $P \hookrightarrow^* Q$ then exists Q' such that $Q \hookrightarrow^* Q'$ with $\mathtt{nf}(P) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(Q'))$.*

Proof: See Appendix A.2. □

Definition 5.9 (Well Formed process) *An HOπ^+ process P is well formed if $(\!|R|\!)l \Rightarrow P$.*

5.4.1 Barbs

In this section we will prove some properties regarding barbs of both $\rho\pi$ configurations and their translations.

Barbs are preserved by the encoding. Formally we have:

Lemma 5.15 *For each consistent configuration M, if $M \downarrow_a$ then $\mathtt{nf}((\!|M|\!)) \downarrow_a$*

Proof: By definition $M \downarrow_a$ implies $M \equiv \nu\tilde{u}.\,(\kappa : a\langle P\rangle \mid N)$ with $a \notin \tilde{u}$. Now $\mathtt{nf}((\!|\nu\tilde{u}.\,(\kappa : a\langle P\rangle \mid N)|\!)) = \nu\tilde{u}.\,\mathtt{nf}((\!|\kappa : a\langle P\rangle \mid N|\!)) = \nu\tilde{u}.\,(\mathtt{nf}((\!|\kappa : a\langle P\rangle|\!)) \mid \mathtt{nf}((\!|N|\!))$. Depending on the value of κ we have two cases: whether $\kappa = k$ or $\kappa = \langle h_i, \tilde{h}\rangle \cdot k$.

In the first case we have that $\nu\tilde{u}.\,(\mathtt{nf}((\!|k : a\langle P\rangle|\!)) \mid \mathtt{nf}((\!|N|\!))) = \nu\tilde{u}.\,(\mathtt{nf}((\!|a\langle P\rangle|\!)k) \mid \mathtt{nf}((\!|N|\!))) = \nu\tilde{u}.\,(a\langle(\!|P|\!), k\rangle \mid \mathtt{nf}(\mathtt{KillM}\ a\ k) \mid \mathtt{nf}((\!|N|\!)))$ and we have that $\nu\tilde{u}.\,(a\langle(\!|P|\!), k\rangle \mid \mathtt{nf}(\mathtt{KillM}\ a\ k) \mid \mathtt{nf}((\!|N|\!))) \downarrow_a$ since $a \notin \tilde{u}$, as desired.

In the second case with $k = \langle h_i, \tilde{h}\rangle \cdot k$ we have $\nu\tilde{u}.\,(\mathtt{nf}((\!|\langle h_i, \tilde{h}\rangle \cdot k : a\langle P\rangle|\!)) \mid \mathtt{nf}((\!|N|\!))) = \nu\tilde{u}.\,(\mathtt{nf}((\!|a\langle P\rangle|\!)h_i) \mid \mathtt{nf}(\mathtt{Kill}_\kappa) \mid \mathtt{nf}((\!|N|\!))) = \nu\tilde{u}.\,(a\langle(\!|P|\!), h_i\rangle \mid \mathtt{nf}(\mathtt{Kill}_\kappa) \mid \mathtt{nf}(\mathtt{KillM}\ a\ h_i) \mid \mathtt{nf}((\!|N|\!)))$ and we have that $\nu\tilde{u}.\,(a\langle(\!|P|\!), h_i\rangle \mid \mathtt{nf}(\mathtt{Kill}_\kappa) \mid \mathtt{nf}(\mathtt{KillM}\ a\ h_i) \mid \mathtt{nf}((\!|N|\!))) \downarrow_a$ since $a \notin \tilde{u}$, as desired. □

The function addG does not add barbs, that is:

Lemma 5.16 *For any HOπ^+process P, if $\mathtt{addG}(P) \downarrow_a$ then $P \downarrow_a$.*

Proof: By simply looking at the Definition 5.4 we can easily see that all the garbage added by the function do not show any barb. □

We cannot state directly that all the barbs shown by $(\!|M|\!)$ are matched by M, since the encoding uses messages on names belonging to $\mathcal{K}$. We then limit the barbs of $(\!|M|\!)$ to those belonging to $\mathcal{N}$, as stated in the following Lemma.

Lemma 5.17 *For each consistent configuration M, if $\mathtt{nf}((\!|M|\!)) \downarrow_a$ with $a \in \mathcal{N}$ then $M \downarrow_a$.*

Proof: By structural induction on M. Since sub terms of well formed configurations are not well formed in general, we have to consider in the induction both well formed configurations and their sub-terms. If $M = \kappa : P$, we proceed by structural induction on P and by case analysis on κ. We will consider just the case in which $\kappa = k$, the other case with $\kappa = \langle h_i, \tilde{h} \rangle \cdot k$ is similar. If $P = 0$ then we have that $\mathtt{nf}(\llparenthesis k : 0 \rrparenthesis) = k\langle \mathtt{Nil} \rangle \mid \mathtt{Rew}\, k$, but since $k \notin \mathcal{N}$ the process $\mathtt{nf}(\llparenthesis k : 0 \rrparenthesis)$ does not show any relevant barb. If $P = a\langle Q \rangle$ then we have that $\mathtt{nf}(\llparenthesis k : a\langle Q \rangle \rrparenthesis) = a\langle \llparenthesis Q \rrparenthesis, k \rangle \mid (a(X, \backslash k) \triangleright k\langle (h)\mathtt{Msg}\; a\; \llparenthesis Q \rrparenthesis\; h \rangle \mid \mathtt{Rew}\;\, k)$, which shows a barb on a. Since also $M \downarrow_a$, we are done. If $P = a(X) \triangleright Q$,we have that $\mathtt{nf}(\llparenthesis k : a(X) \triangleright Q \rrparenthesis) = \nu \mathtt{t}.\, \overline{\mathtt{t}} \mid (a(X, h) | \mathtt{t} \triangleright R) \mid (\mathtt{t} \triangleright S)$ (for some R and S). Since $\mathtt{t}$ is restricted then the entire process does not show any barb, and we are done. If $P = Q_1 \mid Q_2$, the key κ has to be a simple name, since we are dealing with consistent configurations. So, we have that $\mathtt{nf}(\llparenthesis k : (Q_1 \mid Q_2) \rrparenthesis) = \nu h, l.\, \mathtt{nf}(\llparenthesis Q_1 \rrparenthesis h) \mid \mathtt{nf}(\llparenthesis Q_2 \rrparenthesis l) \mid (h(W) | l(Z) \triangleright S)$. The process may show a barb because of either $\mathtt{nf}(\llparenthesis Q_1 \rrparenthesis h)$ or $\mathtt{nf}(\llparenthesis Q_2 \rrparenthesis l)$ (or both). Let us suppose that it is because of $\mathtt{nf}(\llparenthesis Q_1 \rrparenthesis h)$, that is $\mathtt{nf}(\llparenthesis Q_1 \rrparenthesis h) \downarrow_a$. By definition of $\llparenthesis \_ \rrparenthesis$ we have that $\mathtt{nf}(\llparenthesis Q_1 \rrparenthesis h) = \mathtt{nf}(\llparenthesis h : Q_1 \rrparenthesis)$ and hence $\mathtt{nf}(\llparenthesis h : Q_1 \rrparenthesis) \downarrow_a$. Now, by applying the inductive hypothesis we have that $(h : Q_1) \downarrow_a$ and then also $k : (Q_1 \mid Q_2) \downarrow_a$, as desired. The other cases are similar.

If $M = 0$, we have that $\llparenthesis 0 \rrparenthesis = 0$ and the thesis banally follows. If $M = M_1 \mid M_2$ we have that $\mathtt{nf}(\llparenthesis M_1 \mid M_2 \rrparenthesis) = \mathtt{nf}(\llparenthesis M_1 \rrparenthesis \mid \llparenthesis M_2 \rrparenthesis) = \mathtt{nf}(\llparenthesis M_1 \rrparenthesis) \mid \mathtt{nf}(\llparenthesis M_2 \rrparenthesis)$ and we can conclude by applying the inductive hypothesis on $\mathtt{nf}(\llparenthesis M_1 \rrparenthesis)$ and $\mathtt{nf}(\llparenthesis M_2 \rrparenthesis)$. If $M = \nu u.\, M_1$ we have that $\mathtt{nf}(\llparenthesis \nu u.\, M_1 \rrparenthesis) = \nu u.\, \llparenthesis \mathtt{nf}(M_1) \rrparenthesis$ and we can conclude by applying the inductive hypothesis on $\llparenthesis \mathtt{nf}(M_1) \rrparenthesis$. If $M = [\kappa_1 : a\langle P \rangle \mid \kappa_2 : a(X) \triangleright Q; k]$, then $\mathtt{nf}(\llparenthesis M \rrparenthesis) = k(Z) \triangleright R$ (for some R), that shows no barbs and we can conclude. □

As corollary of Lemma 5.17 we have that:

Corollary 5.1 *For any configuration M such that $\nu k.\, k : P \Rightarrow M$, if $\mathtt{nf}(\llparenthesis M \rrparenthesis) \downarrow_a$ then $M \downarrow_a$.*

Lemma 5.18 *For any HOπ^+ process P, if $P \downarrow_a$ and $P \rightharpoondown Q$, then also $Q \downarrow_a$.*

Proof: Directly follows by the definition of $\downarrow_a$ and the application $\rightharpoondown$. □

Barbs are preserved by administrative steps. That is:

Lemma 5.19 *If $M \downarrow_a$ and $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$ then $Q \hookrightarrow^* \downarrow_a$.*

$$\begin{tikzcd}
\llparenthesis M \rrparenthesis \arrow[r, hook, "*"] \arrow[d, harpoon, "*"] \arrow[dr, phantom, "5.11"] & Q \arrow[d, harpoon, "*"] \\
\texttt{nf}(\llparenthesis M \rrparenthesis) \arrow[r, hook, "*"] \arrow[d, phantom, "\equiv_{Ex}"] \arrow[dr, phantom, "5.14"] & Q' \arrow[d, hook, "*"] \\
\texttt{nf}(\texttt{addG}(Q'')) & Q''
\end{tikzcd}$$

Figure 5.6: Correspondence schema of barbs (numbers refer to Lemmas).

Proof: By Lemma 5.15 we have that if $M \downarrow_a$ then $\texttt{nf}(\llparenthesis M \rrparenthesis) \downarrow_a$. By definition of normal form we have that $\llparenthesis M \rrparenthesis \rightharpoondown^* \texttt{nf}(\llparenthesis M \rrparenthesis)$, and by hypothesis we have that $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$. By using Lemma 5.11 we have that $\texttt{nf}(M) \hookrightarrow^* Q'$ and $Q \rightharpoondown^* Q'$. Moreover, by Lemma 5.14 there exists Q'' such that $Q' \hookrightarrow^* Q''$ and $\texttt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q''))$. And since $\equiv_{Ex}$does not remove barbs and $\texttt{addG}$ does not add barbs, we can conclude. The proof is graphically depicted in Figure 5.6.

□

$$\begin{tikzcd}
\llparenthesis M \rrparenthesis \arrow[r, hook, "*"] \arrow[d, harpoon, "*"] \arrow[dr, phantom, "5.11"] & Q \downarrow_a \arrow[d, harpoon, "*"] \\
\texttt{nf}(\llparenthesis M \rrparenthesis) \arrow[r, hook, "*"] & Q' \downarrow_a
\end{tikzcd}$$

Figure 5.7: Barbs with respect to administrative steps (numbers refer to Lemmas).

The following lemma states that starting from a normal form, administrative steps do not add barbs.

Lemma 5.20 *For each consistent configuration M such that $\nu k.\, k : P \Rightarrow M$, if $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$ and $Q \downarrow_a$, then $\texttt{nf}(\llparenthesis M \rrparenthesis) \downarrow_a$.*

Proof: Let us note that $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$ with $Q \downarrow_a$, but also $\llparenthesis M \rrparenthesis \rightharpoondown^* \texttt{nf}(\llparenthesis M \rrparenthesis)$. Now, by applying Lemma 5.11 we have that $Q \rightharpoondown^* Q'$ and $\texttt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* Q'$. Since $Q \downarrow_a$, and since applications ($\rightharpoondown$) do not remove barbs (by Lemma 5.18), we also have that $Q' \downarrow_a$. The above reasoning is depicted in Figure 5.7. Moreover, since $\nu k.\, k : P \Rightarrow M$ then $\texttt{fn}(M) \cap \mathcal{K} = \emptyset$ and also $\texttt{fn}(Q) \cap \mathcal{K} = \emptyset$ (by Lemma A.6). In this way we know that all the messages on channels $l \in \mathcal{K}$ are not barbs.

We now have to show that the barb shown by Q' (and Q) is already present in $\mathtt{nf}(\llparenthesis M \rrparenthesis)$. We proceed by case analysis on the reduction $\hookrightarrow$. Since Q' is generated from $\mathtt{nf}(\llparenthesis M \rrparenthesis)$ via administrative steps, then applications of the form $(((X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ \llparenthesis Q \rrparenthesis\ c)$ or communications of the form $c\langle \llparenthesis P \rrparenthesis \rangle \mid (c(Z) \triangleright Z\ l)$ or of the form $(k(Z) \triangleright (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ l_1) \mid (\mathtt{Trig}\ ((X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ l_2))$ are never enabled. In other words, reductions that may add (weak) barbs are never enabled. So the only communications that can happen are those of killer processes and $\mathtt{Rew}$. Let us note that an internal communication involving a killer does not add barbs (since $\mathtt{fn}(Q') \cap \mathcal{K} = \emptyset$). A communication through a $\mathtt{Rew}$ processes does not add barbs. If the administrative step is an application $\rightharpoondown$ then let us note that all the applications involving killer, $\mathtt{Rew}$ and $\mathtt{Mem}$ processes do not add barbs. All the other applications that may add a barb (such as the application of a $\mathtt{Msg}$ process), are already present in $\mathtt{nf}(\llparenthesis M \rrparenthesis)$ in their applied form. □

Lemma 5.21 *If* $\llparenthesis M \rrparenthesis \hookrightarrow^* \downarrow_a$ *then* $M \downarrow_a$.

Proof: It directly follows by Lemma 5.20 and Corollary 5.1. □

Lemma 5.22 *If* $P \hookrightarrow^* \downarrow_a$ *then* $\mathtt{addG}(P) \hookrightarrow^* \downarrow_a$

Proof: Since $P \hookrightarrow^* P' \downarrow_a$, we can express P as $\mathbb{E}[0]$ and P' as $\mathbb{E}'[0]$ with $\mathbb{E}[0] \hookrightarrow^* \mathbb{E}'[0] \downarrow_a$. Let $\mathtt{addG}(P) = \mathbb{E}[R]$, we still have that $\mathbb{E}[R] \hookrightarrow^* \mathbb{E}'[R] \downarrow_a$, as desired. □

5.4.2 Faithfulness

In order to prove the faithfulness of our encoding (Theorem 5.1) we will prove that the function $\mathtt{addG}$, the structural equivalence $\equiv$ and the structural equivalence $\equiv_{Ex}$ are themselves weak bf barbed bisimulations.

Lemma 5.23 *For any HOπ^+ processes P, Q, the relation $\mathcal{R} = \{(P, Q) \mid P \equiv Q\}$ is a weak bf barbed bisimulation.*

Proof: By induction on the application of $\equiv$ and by case analysis on the last applied axiom. All the cases trivially holds. □

Proposition 5.1 *For any HOπ^+process $P \equiv \nu\tilde{a}.\,P'$, the relation $\mathcal{R} = \{\nu\tilde{a}.\,P',\ \nu\tilde{a}.\,(P' \mid \nu\tilde{b}.\,Q)\}$ with Q a parallel composition of processes as in Definition 5.4 is a weak bf barbed bisimulation.*

Proof: See Appendix A.3. □

Proposition 5.2 *For any HOπ^+process P, Q the relation $\mathcal{R} = \{(P, Q) \mid P \equiv_{Ex} Q\}$ is a weak bf bisimulation.*

Proof: See Appendix A.3. $\square$

If a process is well formed (see Definition 5.9), then all the administrative reductions induced by `addG` can be mimicked by the process itself. That is:

Lemma 5.24 *Let Q be a well formed HOπ^+ process. If* $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow Q'$ *then there exists Q'' such that $Q \hookrightarrow^* Q''$ with $Q' = \texttt{addG}(Q'')$.*

Proof: Let us note that since $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow Q'$, then the reduction is an internal communication and not an application $\rightharpoondown$ (since in the normal form does not contain applications in execution contexts). We have to distinguish two cases: whether the transition $\hookrightarrow$ is due to the process $\texttt{nf}(Q)$ or to the `addG`.

Let us consider the first case. By definition of `addG` we have that $\texttt{addG}(Q) = \mathbb{E}[R]$ with $Q \equiv \mathbb{E}[0]$ where R is the garbage process added by the function. The form of $\texttt{addG}(Q)$ is preserved by the $\texttt{nf}(\cdot)$ function, where all the applications are executed. Hence, $\texttt{nf}(\texttt{addG}(Q)) \equiv \mathbb{E}'[R']$ with $\texttt{nf}(Q) = \mathbb{E}'[0]$. Since the transition is not due to the hole process we have that $\mathbb{E}'[R] \hookrightarrow \mathbb{E}''[R] = Q'$, but also $\mathbb{E}'[0] \hookrightarrow \mathbb{E}''[0] = Q''$. Moreover we have that $Q \rightharpoondown^* \mathbb{E}'[0] \hookrightarrow \mathbb{E}''[0]$ and we are done.

In the second case, note that the only garbage processes that may interact with the context is either a ($\texttt{Rew}\ l$) process or a ($\texttt{KillM}\ a\ l$) process (the other garbage processes are inactive). If the administrative step is due to the applied form of ($\texttt{Rew}\ l$) process this implies that in the context $\mathbb{E}$ is present a message on the channel l, that is $\mathbb{E}[l(Z) \triangleright Z\ l \mid R_1] \equiv \mathbb{E}_1[l\langle S\rangle \mid l(Z) \triangleright Z\ l \mid R_1] \hookrightarrow \mathbb{E}_1[(S\ l) \mid R_1] = Q'$. But since Q is a well formed and since the process $l(Z) \triangleright Z\ l$ has been added by the `addG` function, by Lemma A.11 we have that also $\mathbb{E}_1[l\langle S\rangle] \equiv \mathbb{E}_1[l\langle S\rangle \mid l(Z) \triangleright Z\ l] \hookrightarrow \mathbb{E}_1[(S\ l)] = Q''$. And we can conclude by noting that $Q' = \texttt{nf}(\texttt{addG}(Q''))$. The other case is similar using Lemma A.12 instead of Lemma A.11. $\square$

Lemma 5.25 *Let Q be a well formed HOπ^+ process. If* $\texttt{nf}(\texttt{addG}(Q)) \rightarrow_f Q'$ *then there exists Q'' such that $Q \rightharpoondown^* \rightarrow_f Q''$ with $Q' = \texttt{addG}(Q'')$.*

Proof: It is easy to note that the reduction $\rightarrow_f$ is not induced by the `addG` function, since it only adds processes able to do $\hookrightarrow$ steps. Hence the reduction $\rightarrow_f$ is done by $\texttt{nf}(Q)$ and it is sufficient to chose Q'' such that $Q \hookrightarrow^* \texttt{nf}(Q) \rightarrow_f Q''$ and we are done. $\square$

Lemma 5.26 *Let Q be a well formed HOπ^+ process. If $\mathtt{nf}(\mathtt{addG}(Q)) \rightarrow_b Q'$ then there exists Q'' such that $Q \rightarrow^* \rightarrow_b Q''$ with $Q' = \mathtt{addG}(Q'')$.*

Proof: Similar to the one of Lemma 5.25 □

The following last lemmas show how a forward (backward) transition done by a process generated by our encoding can be matched by a forward (backward) transition of a $\rho\pi$ configuration.

Lemma 5.27 *If $\mathtt{nf}(\llparenthesis M \rrparenthesis) \rightarrow_f P$ then $M \twoheadrightarrow M'$ with $P \hookrightarrow^* P'$ and $\mathtt{nf}(P') \equiv \mathtt{nf}(\mathtt{addG}\llparenthesis M' \rrparenthesis)$.*

Proof: By structural induction on M. If M is a simple process such as $\mathbf{0}$, message process or a trigger there is nothing to verify since $M \not\twoheadrightarrow$.

In the inductive case, if M is a restriction of the form $\nu a.\, M_1$ then by definition of $\mathtt{nf}(\cdot)$ and $\llparenthesis \cdot \rrparenthesis$ we have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) = \nu a.\, \mathtt{nf}(\llparenthesis M_1 \rrparenthesis)$ and by applying the inductive hypothesis on $\mathtt{nf}(\llparenthesis M_1 \rrparenthesis)$ we have that $\llparenthesis \mathtt{nf}(M_1) \rrparenthesis \rightarrow_f P$ implies $M_1 \twoheadrightarrow M_1'$ with $P \hookrightarrow^* P'$ and $\mathtt{nf}(P') = \mathtt{nf}(\mathtt{addG}\llparenthesis M_1' \rrparenthesis)$. We can apply the same reductions on the restriction context, and obtain that also $\nu a.\, \llparenthesis M_1 \rrparenthesis \rightarrow_f \nu a.\, P$ and $\nu a.\, M_1 \twoheadrightarrow \nu a.\, M_1'$ with $\nu a.\, P \hookrightarrow^* \nu a.\, P'$ and since $\mathtt{nf}(P') = \mathtt{nf}(\mathtt{addG}(\llparenthesis M_1' \rrparenthesis))$ we also have that $\nu a.\, \mathtt{nf}(P') = \nu a.\, \mathtt{nf}(\mathtt{addG}(\llparenthesis M_1' \rrparenthesis))$ that is $\mathtt{nf}(\nu a.\, P') = \mathtt{nf}(\mathtt{addG}(\llparenthesis \nu a.\, M_1') \rrparenthesis)$, as desired. The case of parallel context $M = M_1 \mid M_2$ is similar to the restriction one in the case in which the inductive hypothesis is applied on M_1 or M_2, that is the reduction is done inside M_1 or M_2. If both M_1, M_2 contribute to the reduction this means that, let us say, there is a message in M_1 and a trigger in M_2 able to communicate. For the sake of brevity we consider just the case in which both message and trigger are tagged by a simple key. Other cases are similar. So, we can write $M = M_1' \mid k_1 : a\langle R \rangle \mid k_2 : a(X) \triangleright Q \mid M_2'$. Hence, we have

$$\llparenthesis M_1' \mid k_1 : a\langle R \rangle \mid k_2 : a(X) \triangleright Q \mid M_2' \rrparenthesis =$$
$$\llparenthesis M_1' \rrparenthesis \mid \llparenthesis a\langle R \rangle \rrparenthesis k_1 \mid \llparenthesis a(X) \triangleright Q \rrparenthesis k_2 \mid \llparenthesis M_2' \rrparenthesis$$

Let $Y = (X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle$, then we have that

$$\begin{aligned}
&\mathtt{nf}(\llparenthesis M \rrparenthesis) = \mathtt{nf}(\llparenthesis M_1' \rrparenthesis) \mid a\langle \llparenthesis R \rrparenthesis, k_1 \rangle \mid \mathtt{nf}(\mathtt{KillM}\ a\ k_1) \mid \\
&\qquad \nu \mathtt{t}.\, \mathtt{t}(a(X,l) | \mathtt{t} \triangleright_f \nu k, c.\, (Y\ X\ x) \mid (c(Z) \triangleright Z\ k) \mid (\mathtt{Mem}\ Y\ a\ X\ l\ k\ k_2)) \mid \\
&\qquad \mathtt{nf}(\mathtt{KillT}\ Y\ \mathtt{t}\ k_2\ a) \mid \mathtt{nf}(\llparenthesis M_2' \rrparenthesis) \rightarrow_f \\
&\mathtt{nf}(\llparenthesis M_1' \rrparenthesis) \mid \mathtt{nf}(\mathtt{KillM}\ a\ k_1) \mid \nu k, c, \mathtt{t}.\, (Y\ \llparenthesis R \rrparenthesis\ c) \mid (c(Z) \triangleright Z\ k) \mid \\
&(\mathtt{Mem}\ Y\ a\ X\ l\ k\ k_2) \mid \mathtt{nf}(\mathtt{KillT}\ Y\ \mathtt{t}\ k_2\ a) \mid \mathtt{nf}(\llparenthesis M_2' \rrparenthesis) \rightharpoondown \mapsto \\
&\mathtt{nf}(\llparenthesis M_1' \rrparenthesis) \mid \mathtt{nf}(\mathtt{KillM}\ a\ k_1) \mid \nu k, c, \mathtt{t}.\, \llparenthesis Q \rrparenthesis \{^{\llparenthesis R \rrparenthesis}/_X\} k \mid (\mathtt{Mem}\ Y\ a\ \llparenthesis R \rrparenthesis\ k_1\ k\ k_2) \mid \\
&\mathtt{nf}(\mathtt{KillT}\ Y\ \mathtt{t}\ k_2\ a) \mid \mathtt{nf}(\llparenthesis M_2' \rrparenthesis)
\end{aligned}$$

and by using Lemma 5.8 (Substitution Lemma)

$$\begin{aligned}
&\mathtt{nf}(\llparenthesis M_1' \rrparenthesis) \mid \mathtt{nf}(\mathtt{KillM}\ a\ k_1) \mid \nu k, c, \mathtt{t}.\, \llparenthesis Q\{^R/_X\} \rrparenthesis k \mid (\mathtt{Mem}\ Y\ a\ \llparenthesis R \rrparenthesis\ k_1\ k\ k_2) \mid \\
&\mathtt{nf}(\mathtt{KillT}\ Y\ \mathtt{t}\ k_2\ a) \mid \mathtt{nf}(\llparenthesis M_2' \rrparenthesis) = P
\end{aligned}$$

We have that $M \twoheadrightarrow M'$ with $M' = M_1' \mid \nu k.\, k : Q\{^P/_X\} \mid M_2'$ and we can easily see that $P \rightharpoondown^* \mathtt{nf}(\llparenthesis M_1' \rrparenthesis) \mid \mathtt{nf}(\mathtt{KillM}\ a\ k_1) \mid \nu k, c, \mathtt{t}.\, \llparenthesis Q\{^R/_X\} \rrparenthesis k$ $\mid \mathtt{nf}(\mathtt{Mem}\ Y\ a\ \llparenthesis R \rrparenthesis\ k_1\ k\ k_2) \mid \mathtt{nf}(\mathtt{KillT}\ Y\ \mathtt{t}\ k_2\ a) \mid \llparenthesis M_2' \rrparenthesis \equiv \mathtt{nf}(\mathtt{addG}(\llparenthesis M' \rrparenthesis))$, as desired.

□

Lemma 5.28 *Let M be a $\rho\pi$ configuration. If $\llparenthesis M \rrparenthesis \hookrightarrow^* Q \rightarrow_f Q'$ then there are M' and Q'' such that $M \twoheadrightarrow M'$, $\mathtt{addG}(Q') \hookrightarrow^* Q''$ and $\mathtt{nf}(Q'') \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(\llparenthesis M' \rrparenthesis))$.*

Proof: By hypothesis we have that $\llparenthesis M \rrparenthesis \hookrightarrow^* Q \rightarrow_f Q'$ and by definition of normal form and Lemma 5.11 we have that $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$ implies that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* Q_1$ with $Q \rightharpoondown^* Q_1$. Moreover we have that $Q_1 \rightharpoondown^* \mathtt{nf}(Q_1)$ and that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \mathtt{nf}(Q_1)$. Since $Q \rightarrow_f$ and $Q \rightharpoondown^* \mathtt{nf}(Q_1)$ by Lemma 5.11 we have that also $\mathtt{nf}(Q_1) \rightarrow_f$. In order to apply Lemma 5.27 we have to show that also $\mathtt{nf}(\llparenthesis M \rrparenthesis) \rightarrow_f$. We have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \mathtt{nf}(Q_1) \rightarrow_f Q_2$ and we want to show that we can re-arrange the trace $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \mathtt{nf}(Q_1) \rightarrow_f Q_2$ in order to obtain a trace of the form $\mathtt{nf}(\llparenthesis M \rrparenthesis) \rightarrow_f \hookrightarrow^* \mathtt{addG}(Q_2)$. We want to work up-to applications, using Lemma 5.12 we can write $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \mathtt{nf}(Q_1)$ as $\mathtt{nf}(\llparenthesis M \rrparenthesis) \mapsto \rightharpoondown^* \mathtt{nf}(R_1) \mapsto \rightharpoondown^* \ldots \mapsto \rightharpoondown^* \mathtt{nf}(R_n) \mapsto \rightharpoondown^* \mathtt{nf}(Q_1) \rightarrow_f Q_2$.

We now proceed by case analysis on $\mapsto$ starting from the last one. The case in which $\mapsto = \rightarrow_c$ does not apply, since this kind of communication is enabled only by a $\rightarrow_f$ (see Figure 5.3) communication and this communication is not present in the trace $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \mathtt{nf}(Q_1)$. In the case in which $\mapsto$ is a communication due to a $\texttt{Rew}$ process, we have that back in the trace there exists an internal communication due to a killer that produces the message that will be consumed by this $\texttt{Rew}$. This is due to the fact that a $\texttt{Rew}$ process consumes a message on a key channel, and this kind of message is not present in $\texttt{nf}(\llparenthesis M \rrparenthesis)$. Starting from the step $\mapsto$ representing the $\texttt{Rew}$ communication, we go back and select the first (in backward order) $\mapsto$ that produces this message. Since we choose this step to be the first one, this implies that between the step that produces the message and the one that consumes it, there are no steps that consume such a message. Hence we can move the producing step forward in order to be directly followed by the consuming one, and we can then eliminate both the steps obtaining a shorter trace. Let us note that since garbage may be produced by internal communication due to killer process, by eliminating the two steps we obtain a process which may contain less garbage. If the step is due to a killer, it can be postponed after the communication $\rightarrow_f$ since it does not remove processes that contribute to the step $\rightarrow_f$, and we obtain again a shorter trace.

Since $\texttt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \texttt{nf}(Q_1) \rightarrow_f Q_2$ implies $\texttt{nf}(\llparenthesis M \rrparenthesis) \rightarrow_f P \hookrightarrow^* \texttt{addG}(Q_2)$ for some P we can apply Lemma 5.27 and we have that $M \twoheadrightarrow M'$ and $P \hookrightarrow^* P'$ with $\texttt{nf}(P') \equiv_{Ex} \texttt{nf}(\texttt{addG}(\llparenthesis M' \rrparenthesis))$. By using Lemma 5.14 there exists Q_3 such that $\texttt{addG}(Q_2) \hookrightarrow^* Q_3$ with $\texttt{nf}(P) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q_3))$. Since $P \hookrightarrow^* P'$ we have that $\texttt{nf}(P) \hookrightarrow^* \texttt{nf}(P')$ and from Proposition 5.2 we have that $\texttt{nf}(\texttt{addG}(Q_3)) \hookrightarrow^* Q_4 \equiv_{Ex} \texttt{nf}(P')$. We now have $\texttt{addG}(Q_2) \hookrightarrow^* Q_4 \equiv_{Ex} \texttt{nf}(P') \equiv_{Ex} \texttt{nf}(\texttt{addG}(\llparenthesis M' \rrparenthesis))$ as desired. □

Lemma 5.29 *Let M be a $\rho\pi$ configuration. If $\llparenthesis M \rrparenthesis \hookrightarrow^* Q \rightarrow_b Q'$ then there exists M' and Q_2 such that $M \rightsquigarrow M'$, $Q' \hookrightarrow^* Q_2$ and $\texttt{nf}(Q_2) \equiv_{Ex} \texttt{nf}(\texttt{addG}(\llparenthesis M \rrparenthesis))$.*

Proof: We have that $Q \rightarrow_b Q'$ and by applying Lemma 5.11 we also have that $\texttt{nf}(Q) \rightarrow_b Q''$. This implies that $\texttt{nf}(Q) \equiv \nu \tilde{u}.\, R \mid \texttt{nf}(\llparenthesis \texttt{Mem}\ Y\ a\ \llparenthesis P \rrparenthesis\ k_1\ k\ k_2 \rrparenthesis) \mid k\langle \llparenthesis C \rrparenthesis \rangle$ with $Y = (X\ c)c\langle \llparenthesis Q_1 \rrparenthesis \rangle$ and that $Q' \equiv \nu \tilde{u}.\, R \mid (\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ k_1) \mid (\texttt{Trig}\, Y\ a\ k_2)$. Since administrative reductions $\hookrightarrow$ do not remove memories, this implies that the memory is already present in the configuration M and in its normal form. Since $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$ and $\llparenthesis M \rrparenthesis \rightharpoondown^* \texttt{nf}(\llparenthesis M \rrparenthesis)$ by Lemma 5.11 we also have that $\texttt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R$ with $Q \rightharpoondown^* R$, hence $\llparenthesis M \rrparenthesis \hookrightarrow^* R$, moreover since $Q \rightarrow_b Q'$ we also have that $R \rightarrow_b R'$ with $Q' \rightharpoondown^* R'$. By definition of $\llparenthesis \_ \rrparenthesis$ and $\texttt{nf}(\_)$ we have that the process $\texttt{nf}(\llparenthesis M \rrparenthesis)$ does not contain a message on a key channel such as $k\langle \llparenthesis C \rrparenthesis \rangle$ hence this particular message has been generated by the administrative steps in the reduction $\texttt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R$. We have to distinguish two cases: either all the internal communications in $\hookrightarrow^*$ contribute to create such a message or not. In the first case this implies that all the internal communications are due to killer processes, and we have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \equiv \nu\tilde{u}.\, \mathtt{nf}(\llparenthesis N \rrparenthesis) \mid \mathtt{nf}(\mathtt{Mem}\, Y\ a\ \llparenthesis P \rrparenthesis\ k_1\ k\ k_2) \mid \mathtt{nf}(\llparenthesis C \rrparenthesis k)$ for some N, hence $M \equiv \nu\tilde{u}.\, N \mid [k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright Q_1; k] \mid k : C$. Since the administrative steps contribute just in creating the message on the channel k by using Lemma 5.13 (where S is garbage) we have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \nu\tilde{u}.\, \mathtt{nf}(\llparenthesis N \rrparenthesis) \mid (k(Z) \triangleright (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ k_1) \mid (\mathtt{Trig}\ Y\ a\ k_2)) \mid k\langle \llparenthesis C \rrparenthesis \rangle \mid S \rightarrow_b Q''$ with $Q'' = \nu\tilde{u}.\, \mathtt{nf}(\llparenthesis N \rrparenthesis) \mid (\mathtt{Msg}\ a\ \llparenthesis P \rrparenthesis\ k_1) \mid (\mathtt{Trig}\ Y\ a\ k_2)) \mid S$. On the other side we have that $M \rightsquigarrow \nu\tilde{u}.\, N \mid k_1 : a\langle P\rangle \mid k_2 : a(X) \triangleright Q_1 = M'$ and we can note $\mathtt{nf}(Q'') = \mathtt{nf}(\mathtt{addG}(\llparenthesis M' \rrparenthesis))$, as we conclude by choosing $Q_2 = Q''$.

In the case in which there are administrative steps that do not contribute in creating the message on k we do a case analysis on such step. Since we want to re-arrange the trace $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R$ in order to have first all the communication that contribute in creating the message on k followed by all the other unrelated communication, that is $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \hookrightarrow^* R \hookrightarrow R'$, we proceed by natural induction on the length of the reduction $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R$ with a case analysis on the last step that does not contribute to the creation of the message k. It can be either an internal communication due to killer unrelated to the process with tag k, or a communication due to a $\mathtt{Rew}\ l$. In the first case since the kill is on another process not contributing in creating the message on k, that is it is not the kill of the process labelled by k or a kill of a parallel composition due to the split of the key k, it can be postponed, that is if we have $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \hookrightarrow R' \hookrightarrow^* R$, since the step $R_1 \hookrightarrow R'$ is the last one it can be postponed at the end and we can write $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \hookrightarrow^* R \hookrightarrow R'$ and we can conclude by induction on a shorter reduction.

If the step is due to a $\mathtt{Rew}$ process we have two cases: either it deals with processes unrelated to the one tagged by k or not. In the first case we proceed as the case above and we conclude by induction on shorter trace. In the second case, since a communication due to a $\mathtt{Rew}$ instantiates a process, if this process has to be killed in order to form the process in the message k, then there has to be a killer process in order to re-kill it, otherwise this process does not contribute to the creation of the message. Hence, we can annul the effect of these two administrative steps by removing them from the original trace, and we can conclude by induction on a shorter trace.

Once that we have a trace of the form $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \hookrightarrow^* R_2$ with the first trace $\hookrightarrow^*$ containing all the administrative step related to the creation of the message on k, we have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \rightarrow_b R'_1 \hookrightarrow^* R$. As in the first case we know that $\mathtt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* R_1 \rightarrow_f R'_1$ implies that $\llparenthesis M \rrparenthesis \rightsquigarrow \llparenthesis M' \rrparenthesis$ with $\mathtt{nf}(R'_1) = \mathtt{nf}(\mathtt{addG}(\llparenthesis M' \rrparenthesis))$. Moreover we have that $R'_1 \hookrightarrow^* R$ and that $\mathtt{nf}(R'_1) \hookrightarrow^* R''$ with $R \rightharpoondown^* R''$. By using Lemma 5.14 we have that there exist Q_3 such that $R'' \hookrightarrow^* Q_3$ and $\texttt{nf}(Q_3) \equiv_{Ex} \texttt{nf}(\texttt{addG}(R'_1))$ and since $\texttt{nf}(R'_1) = \texttt{nf}(\texttt{addG}(\llparenthesis M' \rrparenthesis))$ we have that $\texttt{nf}(Q_3) \equiv_{Ex} \texttt{nf}(\texttt{addG}(\llparenthesis M' \rrparenthesis))$, as desired. □

We can now prove our main result.

Proof of Theorem 5.1: We prove that the following relation is a weak backward and forward barbed bisimulation:

$$\mathcal{R} = \{(M, N) \mid \nu k.\ k : P \Rightarrow M \wedge \texttt{nf}(N) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q)) \wedge \llparenthesis M \rrparenthesis \hookrightarrow^* Q\}$$

We have to check the different conditions for weak backward and forward barbed bisimulation.

Assume $M \downarrow_a$. Note that from the definition of barbs only names in $\mathcal{N}$ produce barbs. From Lemma 5.19 $Q \hookrightarrow^* \downarrow_a$. Since $\texttt{addG}$ never removes barbs then $\texttt{addG}(Q) \hookrightarrow^* \downarrow_a$. Then also $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow^* \downarrow_a$. Thanks to Proposition 5.2 we have that $\texttt{nf}(N) \hookrightarrow^* \downarrow_a$ and thus also $N \hookrightarrow^* \downarrow_a$.

Assume now $N \downarrow_a$. Thanks to Lemma 5.7 $a \in \mathcal{N}$. We have that $\texttt{nf}(N) \downarrow_a$. Thanks to Proposition 5.2 $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow^* \downarrow_a$ and also $\texttt{addG}(Q) \hookrightarrow^* \downarrow_a$. Then $Q \hookrightarrow^* \downarrow_a$. Finally, thanks to Lemma 5.21 $M \downarrow_a$.

Let us now consider reduction challenges. If $M \to M'$ then by Theorem 5.2 we have $\texttt{nf}(\llparenthesis M \rrparenthesis) \to P$ with $\texttt{nf}(P) \equiv_{Ex} \texttt{nf}(\texttt{addG}(\llparenthesis M' \rrparenthesis))$. By hypothesis we have that $\llparenthesis M \rrparenthesis \hookrightarrow^* Q$, but also $\texttt{nf}(\llparenthesis M \rrparenthesis) \hookrightarrow^* \texttt{nf}(Q)$ and by Lemma 5.14 we have that $\texttt{nf}(Q) \hookrightarrow^* \texttt{nf}(\texttt{addG}(Q')) \equiv_{Ex} \texttt{nf}(\llparenthesis M \rrparenthesis)$. Since $\texttt{nf}(\llparenthesis M \rrparenthesis) \to P$ and $\texttt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q'))$ by Lemma 5.5 we also have that $\texttt{nf}(\texttt{addG}(Q')) \Rightarrow P'$ and $\texttt{nf}(P) \equiv_{Ex} \texttt{nf}(P')$, and since $\texttt{nf}(P) \equiv_{Ex} \texttt{nf}(\texttt{addG}\llparenthesis M' \rrparenthesis)$ by transitivity we have that $\texttt{nf}(P') \equiv_{Ex} \texttt{nf}(\texttt{addG}\llparenthesis M' \rrparenthesis)$. We can conclude by noting that the pair $(M', P') \in \mathcal{R}$.

For the other direction, assume $N \to N'$. We have a case analysis according to the kind of reduction. If $N \rightharpoondown N'$ then the thesis follows trivially since $\texttt{nf}(N) = \texttt{nf}(N')$.

If instead $N \mapsto N'$ then by Lemma 5.11 $\texttt{nf}(N) \hookrightarrow^* N''$ and $N' \rightharpoondown^* N''$ for some N''. Thanks to Lemma 5.12 $\texttt{nf}(N'') = \texttt{nf}(N')$. Thus we have $\texttt{nf}(N) \hookrightarrow^* \texttt{nf}(N')$. Thanks to Proposition 5.2 from $\texttt{nf}(N) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q))$ we have that there is R such that $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow^* R$ and $\texttt{nf}(N') \equiv_{Ex} R$. From $\texttt{nf}(\texttt{addG}(Q)) \hookrightarrow^* R$ we have that $Q \hookrightarrow^* R'$ (thanks to Lemma 5.24) with $\texttt{nf}(R) = \texttt{nf}(\texttt{addG}(R'))$. And we can conclude by using Lemma 5.3 obtaining that $\texttt{nf}(N') \equiv_{Ex} \texttt{nf}(R)$, and hence $\texttt{nf}(N') \equiv_{Ex} \texttt{nf}(\texttt{addG}(R'))$ with $\llparenthesis M \rrparenthesis \hookrightarrow^* Q \hookrightarrow^* R'$ and we remain in the same relation.

Assume now $N \to_f N'$. By Lemma 5.11 $\texttt{nf}(N) \to_f N''$ and $N' \rightharpoondown^* N''$ for some N''. Thanks to Lemma 5.12 $\texttt{nf}(N'') = \texttt{nf}(N')$. Thanks to Proposition 5.2 we have that $\equiv_{Ex}$ is a weak barbed bf bisimulation, and from $\mathtt{nf}(N) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(Q))$ we have that there is R such that $\mathtt{nf}(\mathtt{addG}(Q)) \hookrightarrow^{*} \rightarrow_{f} \hookrightarrow^{*} R$ and $N'' \equiv_{Ex} R$. From Lemma 5.3 also $\mathtt{nf}(N'') = \mathtt{nf}(N') \equiv_{Ex} \mathtt{nf}(R)$. From Lemma 5.24 and Lemma 5.25 we have that there are R', Q_1 and Q_1' such that $Q \hookrightarrow^{*} Q_1 \rightarrow_f Q_1' \hookrightarrow^{*} R'$ and $\mathtt{nf}(R) = \mathtt{nf}(\mathtt{addG}(R'))$. By hypothesis, we have that $\llparenthesis M \rrparenthesis \hookrightarrow^{*} Q$. By Lemma 5.28 we have that $\llparenthesis M \rrparenthesis \hookrightarrow^{*} Q_1 \rightarrow_f Q_1'$ implies that there exist M' and Q_2 such that $M \twoheadrightarrow M'$ and $\mathtt{nf}(\mathtt{addG}\llparenthesis M' \rrparenthesis) \equiv_{Ex} \mathtt{nf}(Q_2)$ with $\mathtt{addG}(Q_1') \hookrightarrow^{*} Q_2$.

We can apply Lemma 5.14 obtaining Q_3 such that $Q_2 \hookrightarrow^{*} Q_3$ and $\mathtt{nf}(Q_3) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(Q_1'))$ and since $Q_1' \hookrightarrow^{*} R'$ we have that $\mathtt{nf}(\mathtt{addG}(Q_1')) \hookrightarrow^{*} \mathtt{nf}(\mathtt{addG}(R')) \equiv_{Ex} \mathtt{nf}(R)$, and since $\mathtt{nf}(N') \equiv_{Ex} \mathtt{nf}(R)$ we also have $\mathtt{nf}(N') \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R'))$. We want to show that the pair $(M', N') \in \mathcal{R}$. We have that $Q_2 \hookrightarrow^{*} Q_3$ but also $\mathtt{nf}(Q_2) \hookrightarrow^{*} \mathtt{nf}(Q_3)$ and since $\mathtt{nf}(\mathtt{addG}\llparenthesis M' \rrparenthesis) \equiv_{Ex} \mathtt{nf}(Q_2)$ and $\equiv_{Ex}$ is a weak bf barbed bisimulation we have that there exists R_2 such that $\mathtt{nf}(\mathtt{addG}\llparenthesis M' \rrparenthesis) \hookrightarrow^{*} R_2$ with $R_2 \equiv_{Ex} \mathtt{nf}(Q_3)$. Hence by using 5.24 we also have that $\mathtt{nf}(\llparenthesis M' \rrparenthesis) \hookrightarrow^{*} R_3$ with $R_2 = \mathtt{addG}(R_3)$. We have $\mathtt{nf}(Q_3) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(Q_1'))$ and $\mathtt{nf}(\mathtt{addG}(Q_1')) \hookrightarrow^{*} \mathtt{nf}(\mathtt{addG}(R'))$ and by the fact that $\equiv_{Ex}$ is a weak bf barbed bisimulation then there exists R_4 such that $\mathtt{nf}(Q_3) \hookrightarrow^{*} R_4$ with $R_4 \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R'))$, but by the same reasoning the have that $R_2 \hookrightarrow^{*} R_5$ with $R_5 \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R'))$. Since $R_2 = \mathtt{addG}(R_3)$ we have that $\mathtt{addG}(R_3) \hookrightarrow^{*} \mathtt{addG}(R_5)$ and by using Lemma 5.4 and Lemma 5.3 we have that $\mathtt{nf}(R_5) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R'))$ and $\mathtt{nf}(\mathtt{addG}(R_5)) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R'))$. To conclude we can note that $\llparenthesis M' \rrparenthesis \rightharpoondown^{*} \mathtt{nf}(\llparenthesis M' \rrparenthesis) \hookrightarrow^{*} \mathtt{addG}(R_2) \hookrightarrow^{*} \mathtt{addG}(R_5)$ with $\mathtt{nf}(\mathtt{addG}(R_5)) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(R')) \equiv_{Ex} \mathtt{nf}(N')$. This implies that $(M', N') \in \mathcal{R}$, as desired.

If $N \rightarrow_b N'$ we can use the same reasoning of the $\rightarrow_f$ case by using Lemma 5.29 and Lemma 5.26 instead of Lemma 5.28 and Lemma 5.25. $\square$

5.5 roll-π encoding

In this section we show how the encoding of $\rho\pi$ (in Figure 5.4 and Figure 5.3) can be used, with a small change, to encode a variant of the roll-π low level semantics of Section 4.5. Before introducing the new encoding, we need to formalize a few functions that will be used by the encoding.

Definition 5.10 *Let $\phi : \mathcal{K} \to \mathcal{K} \times \mathcal{K}$ be a function defined as follows:*

$$\forall h, k \in \mathcal{K} \quad h \neq k \Longrightarrow \{h_1, h_2\} \cap \{k_1, k_2\} = \emptyset$$

where $\phi(h) = (h_1, h_2)$ and $\phi(k) = (k_1, k_2)$.

Definition 5.11 *We define two projection functions $\pi_1, \pi_2 : \mathcal{K} \to \mathcal{K}$ such that:*

$$\phi(k) = (k_1, k_2) \quad \Longleftrightarrow \quad \pi_1(k) = k_1 \quad \pi_2(k) = k_2$$

Function ϕ defines an injective mapping from a single key to a pair, and this mapping is unique that is two different keys are mapped in two different pairs. Moreover functions $\pi_1(k), \pi_2(k)$ can be seen respectively as the projection of the first and the second element of $\phi(k)$.

Notations and conventions. Abusing the HOπ^+ notation, we use the bold name $\mathbf{l}$ to indicate the pair l_1, l_2. Hence abstractions over two channels instead of being written as $(l_1\ l_2)P$ will be written as $(\mathbf{l})P$. We will use the same trick also for applications, that is instead of having $(P\ l_1\ l_2)$, where P is an abstraction, we sometimes will write $(P\ \mathbf{l})$. Moreover if P is an abstraction over two channels, sometimes we will write $(P\ \phi(k))$ instead of writing $(P\ \pi_1(k)\ \pi_2(k))$ for any $k \in \mathcal{K}$. Other conventions and notations are similar to the encoding of $\rho\pi$.

The encoding $(\!|\cdot|\!) : \mathcal{P}_{\text{roll-}\pi} \to \mathcal{P}_{\text{HO}\pi^+}$ of processes of roll-π in HOπ^+ is defined inductively in Figure 5.8. It extends to an encoding $(\!|\cdot|\!) : \mathcal{C}_{\text{roll-}\pi} \to \mathcal{P}_{\text{HO}\pi^+}$ of configurations of roll-π in HOπ^+ as given in Figure 5.9 (note that the encoding for $\mathbf{0}$ in Figure 5.4 is the encoding for the null configuration). The main idea behind this encoding is that a roll-π tag key $k \in \mathcal{K}$ is interpreted as a pair of channels. Hence the encoding of processes is an abstraction over these two channels. The first channel is used by a process to be *notified* by a roll notification to start rolling-back, while the second channel is used by the process to freeze itself and notify its father that it has rolled-back. Naturally, as in the LL semantics, a process that receives a notification and is not a leaf (a primitive process) will propagate the notification through its children. So this encoding mimics the two way visit of the tree that we have discussed in Chapter 4.

Let us start commenting on the encoding of roll-π configurations. The null configuration **0** is trivially translated into the HOπ^+ process **0**. Name creation is translated into a name creation. Since we are assuming that there exists a unique correspondence from roll-π keys to pairs of keys, then the roll-π name creation is translated into the creation of the two corresponding keys given by the mapping ϕ. The encoding of a tagged process is similar to the one of $\rho\pi$ with the slight difference that now translations are abstractions over two keys, and these keys are given by the two projections π_1 and π_2. We will discuss the encoding of memories later on, while introducing the encoding of roll-π processes. A frozen process of the form $\lfloor k : P \rfloor$ is encoded as a message on the second key channel carrying the encoding of the process itself, that is:

$$\lfloor k : P \rfloor = k_2\langle (\!| P |\!) \rangle \quad \text{if} \quad \pi_2(k) = k_2$$

As already said, this is one of the intuitions behind this encoding: the second key channel is used to represent a frozen process. The encoding of a frozen process tagged by a complex key $\langle h_i, \tilde{h} \rangle \cdot k$ is pretty similar, where as in the encoding of $\rho\pi$ we use the channel h_i as the process key channel. That is:

$$\lfloor \langle h_i, \tilde{h} \rangle \cdot k : P \rfloor = h_i'' \langle (\!| P |\!) \rangle \quad \text{if} \quad \pi_2(h_i) = h_i''$$

A runtime notification $\mathsf{rl}\ k$ is encoded as a signal (empty message) on the first key channel. That is:

$$(\!| \mathsf{rl}\ k |\!) = \overline{k_1} \quad \text{if} \quad \pi_1(k) = k_1$$

This is the second intuition behind this encoding: the first channel of the mapping is used by processes to receive notifications about rollback.

Let us now comment on the encoding of processes. Since now the rollback facility is controlled, then there is no need of `Rew` processes (as in the $\rho\pi$ encoding) in order to undo a rollback decision. Moreover, now a killer process will await a signal on the first key channel (of the pair), before rolling-back a process. A process of the form $k : 0$ is encoded as follows[5]:

$$(\!| k : 0 |\!) = k_1 \triangleright k_2\langle \mathtt{Nil} \rangle \quad \text{if} \quad \phi(k) = (k_1, k_2)$$

Simply the 0 process awaits the notification, on the first channel, to roll-back

[5] After several applications.

and then produces a message on the second key channel containing itself. Messages now are translated into tri-adic messages. Let us note that the process `KillM`:

$$a(X, \backslash l_1, \backslash l_2)|l_1 \triangleright l_2\langle(\mathbf{h})\texttt{Msg}\, a\, X\, \mathbf{h}\rangle)$$

in order to rollback a particular message on a awaits also a notification on the first key channel l_1. Then the message is rolled-back into the second key channel l_2. The translation of a parallel process is quite straightforward: two new pairs of names are created and given to the two sub-processes composing the parallel process, and to its killer. Let us comment on the `KillP` process:

$$(l_1 \triangleright (h_2(W) \mid k_2(Z) \triangleright l_2\langle(\mathbf{l})\texttt{Par}\, W\, Z\, \mathbf{l}\rangle) \mid \overline{h_1} \mid \overline{k_1})$$

We can see how the notification propagation mechanism works: the process awaits on its first key channel l_1 the notification, then it propagates this notification to its children by means of the two messages $\overline{h_1}$ and $\overline{k_1}$, and awaits the rollback of the children that will be notified on channels h_2 and k_2. When the children have rolled-back the process rolls-back the entire parallel composition by just recomposing itself on the channel l_2.

The translation of a trigger process is similar to the $\rho\pi$ encoding with the slight difference that instead of creating a new key, we create two keys and that the `KillT` process awaits the roll notification (and the lock $\overline{\texttt{t}}$) in order to roll-back the trigger process. Another difference with the $\rho\pi$ encoding is that now triggers are also binders for tag variables γ. Therefore, the translation of a trigger $l : a(X) \triangleright_\gamma Q$ is a process of the form:

$$\nu\texttt{t}.\, \overline{\texttt{t}} \mid (a(X, h_1, h_2) \mid \texttt{t} \triangleright_f \nu\mathbf{k}, c.\, (Y\ X\ \mathbf{k}\ c) \mid (c(Z) \triangleright (Z\ \mathbf{k})) \mid$$

$$(\texttt{Mem}\, Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l})) \mid (\texttt{KillT}\, Y\ \texttt{t}\ \mathbf{l}\ a)$$

with $Y = (X\ \gamma_1\ \gamma_2\ c)c\langle(\!|Q|\!)\rangle$. Let us note that now the process Y is also an abstraction on two channels representing the γ variable of the trigger. The translation of the trigger mimics exactly the roll-π forward rule: it creates the new key k, it substitutes the variable X with the message content and the new key with the γ variable. For example, let us suppose that after the communication we obtain the following process: $k : Q\{^{P,k}/_{X,\gamma}\}$, with Q the trigger body. This substitution is mimicked in our encoding by the following reduction:

$$((X\ \gamma_1\ \gamma_2\ c)c\langle(\!|Q|\!)\rangle)\ (\!|P|\!)\ k_1\ k_2\ c \to c\langle(\!|Q|\!)\rangle\{^{k_1,k_2,(\!|P|\!)}/_{\gamma_1,\gamma_2,X}\}$$

with $\phi(k) = (k_1, k_2)$. The trigger continuation, resulting from the substitution, is then applied to the pair of keys corresponding to the new key k (by the sub-process $c(Z) \triangleright Z\ k_1\ k_2$) and eventually a memory process $\texttt{Mem}$ is created.

$\texttt{Mem}$ processes are quite different from the $\rho\pi$ encoding. A $\texttt{Mem}$ process of the form:

$$\nu \texttt{t}. \overline{\texttt{t}} \mid (k_2(Z)|\texttt{t} \triangleright_b (\texttt{Msg}\ a\ X\ \mathbf{h}) \mid (\texttt{Trig}\ Y\ a\ \mathbf{l})) \mid$$

$$(h_1|\texttt{t} \triangleright (\texttt{Mem}_{\texttt{sx}}\ Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l}) \mid \overline{k_1}) \mid (l_1|\texttt{t} \triangleright \texttt{Mem}_{\texttt{dx}}\ Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l}) \mid \overline{k_1})$$

can release its content because its continuation has fully rolled-back, and this rollback is represented by the message on the key channel k_2. Otherwise, it can propagate the rollback notification to its continuation. That is, if there exists a notification for its internal output process (represented by a signal $\overline{h_1}$) then the memory freezes the internal output process and produces the notification $\overline{k_1}$ in order to roll-back its continuation. Eventually when the continuation will be frozen, then the memory will release its content represented by a frozen message and a trigger, as part of the process $\texttt{Mem}_{\texttt{sx}}$. On the other hand, if the memory receives a notification trying to freeze its internal trigger (signal $\overline{l_1}$) then it propagates the notification to its continuation and behaves as the process $\texttt{Mem}_{\texttt{dx}}$, representing a memory whose internal trigger is frozen. We can note that notification propagation is performed just by $\texttt{KillP}$ and $\texttt{Mem}$ processes, representing nodes in the causal tree.

Since the encoding is defined inductively on the structure of processes we have two kinds of roll process: one in which the tag variable γ is already instantiated and the other where it is not. A process $k : \mathsf{roll}\ l$ is encoded as follows:

$$(k_1 \triangleright k_2\langle \texttt{Roll}\rangle) \mid \overline{l_1} \quad \text{if} \quad \phi(k) = (k_1, k_2) \ \wedge \ \phi(l) = (l_1, l_2)$$

A roll process produces the notification to the target memory (via the signal $\overline{l_1}$) and awaits a notification (from its father) to roll-back. So, as in the LL semantics, a rollback operation in the encoding is also initiated by a roll process.

$$
\begin{array}{ll}
\llparenthesis \mathbf{0} \rrparenthesis = \texttt{Nil} & \llparenthesis X \rrparenthesis = X \\
\llparenthesis a\langle P\rangle \rrparenthesis = (\mathbf{l})(\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ \mathbf{l}) & \llparenthesis \nu a.\, P \rrparenthesis = (\mathbf{l})\nu a.\, \llparenthesis P \rrparenthesis\, \mathbf{l} \\
\llparenthesis P \mid Q \rrparenthesis = (\mathbf{l})(\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ \mathbf{l}) & \llparenthesis a(X) \triangleright_{\gamma} P \rrparenthesis = (\mathbf{l})(\texttt{Trig}\ ((X\ \gamma_1\ \gamma_2\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ \mathbf{l}) \\
\llparenthesis \textsf{roll}\ \mathbf{k} \rrparenthesis = (\mathbf{l})(\texttt{Roll}\ \mathbf{l}\ \mathbf{k}) & \llparenthesis \textsf{roll}\ \gamma \rrparenthesis = (\mathbf{l})(\texttt{Roll}\ \mathbf{l}\ \gamma_1\ \gamma_2) \\
\llparenthesis \dagger \rrparenthesis = (\mathbf{l})(l_1 \triangleright l_2\langle \texttt{Roll} \rangle) &
\end{array}
$$

$$
\begin{aligned}
&\texttt{Nil} = (\mathbf{l})l_1 \triangleright l_2\langle \texttt{Nil} \rangle \\
&\texttt{Roll} = (\mathbf{l}\ \mathbf{k})((l_1 \triangleright l_2\langle \texttt{Roll} \rangle) \mid \overline{k_1}) \\
&\texttt{Msg} = (a\ X\ l_1\ l_2)a\langle X, \mathbf{l} \rangle \mid (\texttt{KillM}\, a\ l_1\ l_2) \\
&\texttt{KillM} = (a\ l_1\ l_2)(a(X, \backslash l_1, \backslash l_2)| l_1 \triangleright l_2\langle (\mathbf{h})\texttt{Msg}\, a\, X\, \mathbf{h} \rangle) \\
&\texttt{Par} = (X\ Y\ l_1\ l_2)\nu h_1, h_2, k_1, k_2.\, X\, \mathbf{h} \mid Y\, \mathbf{k} \mid (\texttt{KillP}\ \mathbf{h}\ \mathbf{k}\ \mathbf{l}) \\
&\texttt{KillP} = (\mathbf{h}\ \mathbf{k}\ \mathbf{l})(l_1 \triangleright (h_2(W) \mid k_2(Z) \triangleright l_2\langle (\mathbf{l})\texttt{Par}\, W\ Z\, \mathbf{l} \rangle) \mid \overline{h_1} \mid \overline{k_1}) \\
&\texttt{Trig} = (Y\ a\ \mathbf{l})\nu \texttt{t}.\, \overline{\texttt{t}} \mid (a(X, \mathbf{h}) \mid \texttt{t} \triangleright_f \nu \mathbf{k}, c.\, (Y\ X\ \mathbf{k}\ c) \mid (c(Z) \triangleright (Z\ \mathbf{k})) \mid \\
&\qquad\quad (\texttt{Mem}\, Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l})) \mid (\texttt{KillT}\, Y\ \texttt{t}\ \mathbf{l}\ a) \\
&\texttt{KillT} = (Y\ \texttt{t}\ \mathbf{l}\ a)(\texttt{t} \mid l_1 \triangleright l_2\langle (\mathbf{h})\texttt{Trig}\, Y\, a\, \mathbf{h} \rangle) \\
&\texttt{Mem} = (\ Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l})\nu \texttt{t}.\, \overline{\texttt{t}} \mid (k_2(Z)| \texttt{t} \triangleright_b (\texttt{Msg}\ a\ X\ \mathbf{h}) \mid (\texttt{Trig}\, Y\ a\ \mathbf{l})) \mid \\
&\qquad\quad (h_1| \texttt{t} \triangleright (\texttt{Mem}_{\texttt{sx}}\ Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l}) \mid \overline{k_1}) \mid (l_1| \texttt{t} \triangleright \texttt{Mem}_{\texttt{dx}}\ Y\ a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l}) \mid \overline{k_1}) \\
&\texttt{Mem}_{\texttt{sx}} = (\ Y a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l})(k_2(Z) \triangleright h_2\langle (\mathbf{h})(\texttt{Msg}\ a\ X\ \mathbf{h}) \rangle \mid (\texttt{Trig}\, Y\ a\ \mathbf{l})) \\
&\texttt{Mem}_{\texttt{dx}} = (\ Y a\ X\ \mathbf{h}\ \mathbf{k}\ \mathbf{l})(k_2(Z) \triangleright (\texttt{Msg}\ a\ X\ \mathbf{h}) \mid l_2\langle (\mathbf{l})(\texttt{Trig}\, Y\ a\ \mathbf{l}) \rangle)
\end{aligned}
$$

Figure 5.8: Encoding roll-π processes.

5.5.1 Correctness

We now conjecture a faithfulness property of the encoding with respect to roll-π.

Conjecture 5.1 *For each HL consistent configuration M, $M \overset{\circ}{\approx}_c \llparenthesis M \rrparenthesis$.*

A possible proof strategy in order to prove Conjecture 5.1, is to first define the closest semantics (called ALL for asynchronous low level) to the encoding and then to relate it with the LL one. The ALL semantics $\rightarrow_{ALL}$ of roll-π is defined as for the HL one (cf. Section 3.2.1), as $\rightarrow_{ALL} = \twoheadrightarrow_{ALL} \cup \rightsquigarrow_{ALL}$, where relations $\twoheadrightarrow_{ALL}$ and $\rightsquigarrow_{ALL}$ are defined to be the smallest evaluation-closed binary relations on closed ALL configurations satisfying the rules in Figure 5.10. The notion of structural congruence used in the definition of evaluation-closed is here the smallest congruence on ALL processes and configurations that satisfies the rules in Figure 3.2 and in Figure 5.11.

$$
\begin{aligned}
&\llparenthesis \mathbf{0} \rrparenthesis = \mathbf{0}\\
&\llparenthesis M \mid N \rrparenthesis = \llparenthesis M \rrparenthesis \mid \llparenthesis N \rrparenthesis\\
&\llparenthesis \nu a.\, M \rrparenthesis = \nu a.\, \llparenthesis M \rrparenthesis\\
&\llparenthesis \nu k.\, M \rrparenthesis = \nu \phi(k).\, \llparenthesis M \rrparenthesis\\
&\llparenthesis k : P \rrparenthesis = (\llparenthesis P \rrparenthesis\ \pi_1(k)\ \pi_2(k))\\
&\llparenthesis \langle h_i, \tilde{h} \rangle \cdot k : P \rrparenthesis = (\llparenthesis P \rrparenthesis\ \pi_1(h_i)\ \pi_2(h_i)) \mid \mathtt{Kill}_{\langle h_i, \tilde{h} \rangle \cdot k}\\
&\llparenthesis [\kappa_1 : a\langle P \rangle \mid \kappa_2 : a(X) \triangleright Q; k] \rrparenthesis =\\
&\qquad (\mathtt{Mem}\ ((X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)\ a\ \llparenthesis P \rrparenthesis\ \llparenthesis \kappa_1 \rrparenthesis\ \pi_1(k)\ \pi_2(k)\ \llparenthesis \kappa_2 \rrparenthesis) \mid \mathtt{Kill}_{\kappa_1} \mid \mathtt{Kill}_{\kappa_2}\\
&\llparenthesis [\lfloor \kappa_1 : a\langle P \rangle \rfloor \mid \kappa_2 : a(X) \triangleright_\gamma Q; k] \rrparenthesis =\\
&\qquad (\mathtt{Mem}_{\mathtt{sx}}\ ((X\ \gamma_1\ \gamma_2\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)\ a\ \llparenthesis P \rrparenthesis\ \llparenthesis \kappa_1 \rrparenthesis\ \pi_1(k)\ \pi_2(k)\ \llparenthesis \kappa_2 \rrparenthesis) \mid \mathtt{Kill}_{\kappa_2}\\
&\llparenthesis [\kappa_1 : a\langle P \rangle \mid \lfloor \kappa_2 : a(X) \triangleright_\gamma Q \rfloor; k] \rrparenthesis =\\
&\qquad (\mathtt{Mem}_{\mathtt{dx}}\ ((X\ \gamma_1\ \gamma_2\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)\ a\ \llparenthesis P \rrparenthesis\ \llparenthesis \kappa_1 \rrparenthesis\ \pi_1(k)\ \pi_2(k)\ \llparenthesis \kappa_2 \rrparenthesis) \mid \mathtt{Kill}_{\kappa_1}\\
&\llparenthesis \lfloor k : P \rfloor \rrparenthesis = k_2\langle \llparenthesis P \rrparenthesis \rangle \ \text{ if } \ \pi_2(k) = k_2\\
&\llparenthesis \lfloor \langle h_i, \tilde{h} \rangle \cdot k : P \rfloor \rrparenthesis = h'\langle \llparenthesis P \rrparenthesis \rangle \ \text{ if } \ \pi_2(h_i) = h'\\
&\llparenthesis \mathsf{rl}\ k \rrparenthesis = \overline{k_1} \ \text{ if } \ \pi_1(k) = k_1\\
&\llparenthesis \mathsf{rl}\ \langle h_i, \tilde{h} \rangle \cdot k \rrparenthesis = \overline{h'} \ \text{ if } \ \pi_1(h_i) = h'\\
&\llparenthesis k \rrparenthesis = \phi(k)\\
&\llparenthesis \langle h_i, \tilde{h} \rangle \cdot k \rrparenthesis = \phi(h_i)
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{Kill}_{\langle h_1, \tilde{h} \rangle \cdot k} = \nu \tilde{l}.\, &(\mathtt{KillP}\ \phi(h_1)\ \phi(l_1)\ \phi(k)) \mid \prod_{i=2}^{n-2} (\mathtt{KillP}\ \phi(h_i)\ \phi(l_i)\ \phi(l_{i-1})) \mid\\
&(\mathtt{KillP}\ \phi(h_{n-1})\ \phi(h_n)\ \phi(l_{n-2}))\\
\mathtt{Kill}_{\kappa} = \mathbf{0} \quad &\text{otherwise}
\end{aligned}
$$

Figure 5.9: Encoding roll-π configurations.

$$\text{(A.}\textsc{Com}\text{)}\qquad \frac{\mu = (\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q)}{(\kappa_1 : a\langle P\rangle) \mid (\kappa_2 : a(X) \triangleright_\gamma Q) \twoheadrightarrow_{ALL} \nu k.\, (k : Q\{{}^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(A.}\textsc{Start}\text{)}\quad (\kappa_1 : \mathsf{roll}\ k) \rightsquigarrow (\kappa_1 : \dagger) \mid \mathsf{rl}\ k$$

$$\text{(A.}\textsc{Span}\text{)}\quad \mathsf{rl}\ \kappa_1 \mid [\kappa_1 : P \mid \kappa_2 : Q; k] \rightsquigarrow_{ALL} [\lfloor \kappa_1 : P \rfloor \mid \kappa_2 : Q; k] \mid \mathsf{rl}\ k$$

$$\text{(A.}\textsc{Branch}\text{)}\quad \frac{\langle h_i, \tilde{h}\rangle \cdot k \text{ occurs in } M}{\mathsf{rl}\ k \mid M \rightsquigarrow_{ALL} \prod_{h_i \in \tilde{h}} \mathsf{rl}\ \langle h_i, \tilde{h}\rangle \cdot k \mid M}$$

$$\text{(A.}\textsc{Up}\text{)}\quad \mathsf{rl}\ \kappa_1 \mid (\kappa_1 : P) \rightsquigarrow_{ALL} \lfloor \kappa_1 : P \rfloor$$

$$\text{(A.}\textsc{Stop}\text{)}\quad [\mu; k] \mid \lfloor k : P \rfloor \rightsquigarrow_{ALL} \mu$$

Figure 5.10: Reduction rules for ALL.

$$\text{(E.}\textsc{Gb}\text{)}\quad \nu k.\, \prod_{i \in I} \mathsf{rl}\ \kappa_i \equiv_{LL} 0 \quad k \subset \kappa_i$$

$$\text{(E.}\textsc{TagPFr}\text{)}\ \lfloor k : \prod_{i=1}^{n} \tau_i \rfloor \equiv_{LL} \nu \tilde{h}.\, \prod_{i=1}^{n} \lfloor \langle h_i, \tilde{h}\rangle \cdot k : \tau_i \rfloor \quad \tilde{h} = \{h_1, \ldots, h_n\} \quad n \geq 2$$

Figure 5.11: Additional structural laws for ALL.

Let us comment on the semantics. Communication (forward) rule is similar to all the given semantics. Rule A.Start generates *asynchronously* a runtime roll notification. In order to avoid an unbounded number of notifications from the same roll process, a roll process that generates a notification becomes the *dummy* process †. For consistency reasons, we let the dummy process † bearing the same tag of the roll process instead of **0**. Rule A.Span behaves as the rule L.Span (of the *LL* semantics) with the slight difference that it is possible to apply the rule just on memories that do not contain frozen processes. Rules A.Branch, A.Up and A.Stop are equivalent to the corresponding ones of the *LL* semantics. Note that, since rl notifications are generated asynchronously there is no more need of marked memories ($[\mu; k]^{\bullet}$). Structural laws are equivalent to those of *LL* semantics.

We now conjecture that an *ALL* configuration and its translation are weak bf barbed congruent, and that the *LL* and *ALL* semantics are equivalent by means of ${}_{LL}\dot{\approx}^{c}_{ALL}$ (see Definition 4.7).

Conjecture 5.2 *For each ALL consistent configuration M, $M \overset{\circ}{\approx}_c (\!|M|\!)$*

Conjecture 5.3 *For each LL consistent configuration M, $M\ {}_{LL}\dot{\approx}^{c}_{ALL}\ (\!|M|\!)$.*

Assuming that the above conjectures hold, we can prove Conjecture 5.1 as follows:

Proof sketch: By Theorem 4.7 we have that $M\ {}_{HL}\dot{\approx}^{c}_{LL}\ M$ and by Conjecture 5.3 and by transitivity we have that $M\ {}_{HL}\dot{\approx}^{c}_{ALL}\ M$. By Conjecture 5.2 we have that $M \overset{\circ}{\approx}_c (\!|M|\!)$ and by transitivity we can conclude that $M \overset{\circ}{\approx}_c (\!|M|\!)$, as desired. □

Chapter 6

Conclusion

6.1 Concluding Remarks

We have proposed in this thesis the Reversible Higher-Order π-calculus ($\rho\pi$), a reversible variant of the Higher-Order π (HOπ). Following the study undertaken by Danos and Krivine on reversible CCS [29, 30], we devised a simple syntax and reduction semantics for $\rho\pi$, with a novel way of defining reversible reductions that preserves the associativity and commutativity of the parallel operator. As far as we know, $\rho\pi$ is the first reversible higher-order concurrent calculus. We have proved that $\rho\pi$ is a conservative extension of HOπ, as long as forward executions are considered, and we have proved that in the calculus the reversible actions are causally consistent.

In $\rho\pi$ each process moves freely backward and forward. The notion of memory introduced in $\rho\pi$ is in some way a checkpoint, uniquely identified by its tag. We exploited this intuition to introduce an explicit form of backward reduction in another calculus: roll-π. In roll-π backward reduction is not allowed by default as in $\rho\pi$, but is triggered by an instruction of the form roll k, whose intent is that the current computation be rolled-back to a state just prior to the creation of the memory bearing the tag k. The definition of a proper semantics for such a primitive is a delicate matter because of the potential interferences between concurrent rollbacks. We have defined in this thesis a high-level operational semantics for roll-π, which we have proved to be sound and complete with respect to $\rho\pi$ backward reduction. We also have defined a lower-level distributed implementation of such a primitive, closer to an actual distributed algorithm, and we have proved it to be fully abstract with respect to the high-level semantics.

The last question we have answered is whether reversibility ($\rho\pi$) can be directly expressed in HOπ. We have shown how to encode $\rho\pi$ into a variant of HOπ that we call HOπ^+. This variant allows the use of join patterns, sub-addressing and abstractions. All these constructs are well known and well understood in terms of expressive power with respect to HOπ. We have proved the faithfulness of our encoding by means of a behavioural equivalence. Moreover, with a simple modification to the encoding, we have obtained an encoding of roll-π into HOπ^+, whose correctness is just conjectured.

6.2 Ongoing and Future work

Starting from the work presented in this thesis we see four main directions of future research: semantics for reversible concurrent processes, programming abstractions for dependable reversible systems, implementation and expressiveness (of reversible actions) issues. Let us detail them.

Semantics. The study of behavioral equivalences for concurrent processes typically involves the definition of contextual equivalences and their coinductive characterisation, in terms of labelled transition system (LTS) and bisimulations, that provide a mathematically appealing technique for proving the equivalence of processes. As emerged in Section 3.4, we do not yet understand what is a correct notion of behavioral equivalence for reversible processes. For instance, our early attempt in [58] at defining a weak notion of barbed congruence (a classical notion of contextual equivalence for concurrent languages [73, 89]) for a reversible variant of the Higher-Order π-calculus proved to be insufficiently discriminating. Besides our notion of backward and forward barbed bisimulation used in this thesis for the faithfulness property of our encoding, is an ad-hoc equivalence and it is not a candidate to be the canonical one. This leads us to think that this is a non-trivial issue; even more perplexing seems to be the definition of appropriate coinductive characterizations for these more discriminating variants. Intriguingly, the strong counterpart of barbed congruence appears to be less problematic, and previously studied notions of back and forth bisimulations, e.g. [33, 51, 68] seem to be good candidates for its coinductive characterization. Note however that previous work on back-and-forth bisimulations focused on exploiting backward steps as an auxiliary tool for better understanding purely forward computations, and on the study of strong equivalences.

The notion of reversibility in concurrent systems has a tight relationship with causality. This is very apparent from our $\rho\pi$, where the structure of process tags and memories appears to capture causality information in a way very much related to works that investigate a causal semantics for concurrent process calculi [16, 25, 27, 96]. A thorough analysis of this relationship must be left for further study.

Following the work done by Phillips and Ulidowski [82], we could generalize our reversing approach by proposing a general framework to reverse calculi with binders and higher-order aspects expressed for example in the SOS format of [77].

Programming Abstractions. In this thesis we have assumed that all the actions in our system can be reversible. In a real world, and in distributed systems, it is unthinkable to consider every action reversible. There are actions that by nature are irreversible, such as printing a file. Hence, it would be interesting to extend $\rho\pi$ with a notion of irreversible actions, that cannot be reversed but compensated. So it could be interesting to mix reversibility and mechanisms for dynamic compensations such as those proposed in [60, 74].

In Chapter 4 we presented a low level implementation of our controlled reversible facility. Our low-level semantics for rollback, being a first refinement towards an implementation, is certainly related to distributed checkpoint and rollback schemes, in particular to the causal logging schemes discussed in the survey [37]. A thorough analysis of this relationship must be left for further study, however, as it requires a proper modelling of site and communication failures (where sites can be expressed by localities), as well as an explicit model for persistent data.

The way of controlling $\rho\pi$ reversibility, presented in Chapter 4, is just one among several. Another direction will be to study other controlling primitives, such as the one typical of reversible debuggers, allowing to go backward n computational steps, and to compare them.

Implementation. Our ultimate goal is to provide a programming language for building reliable distributed systems by means of backward recovery techniques. An interesting research question is to extend a lambda calculus for concurrent systems (such as the one presented in [17]) or a functional programming language (such as OCaml) with reversible actions, and see what is the actual cost in terms of memory consumption of our reversible mechanism.

Expressiveness. Another research direction is to refine the $\rho\pi$ encoding presented in Chapter 5, in order to avoid (if possible) the generation of garbage processes and try to limit divergence.

Bibliography

[1] Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WSBPEL) TC, Apr. 2007.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 63–74, New York, NY, USA, 2008. ACM.

[3] M. Abadi and T. Harris. Perspectives on Transactional Memory. In *20th International Conference on Concurrency Theory (CONCUR)*, volume 5710 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2009.

[4] S. Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347(3):441–464, 2005.

[5] L. Acciai, M. Boreale, and S. Dal Zilio. A Concurrent Calculus with Atomic Transactions. In *16th European Symposium on Programming (ESOP)*, volume 4421 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2007.

[6] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *ACM Trans. Softw. Eng. Methodol.*, 13(2):149–198, 2004.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[8] H. B. Axelsen and R. Glück. What Do Reversible Programs Compute? In *FOSSACS*, pages 42–56, 2011.

[9] H. B. Axelsen, R. Glück, and T. Yokoyama.
Reversible Machine Code and Its Abstract Processor Architecture.
In *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*, pages 56–69, 2007.

[10] H. Barendregt.
The Lambda Calculus – Its Syntax and Semantics.
North-Holland, revised edition, 1984.

[11] P. Benioff.
Quantum Mechanical Models of Turing Machines That Dissipate No Energy.
Phys. Rev. Lett., 48:1581–1585, 1982.

[12] C. Bennett.
Logical Reversibility of Computation.
IBM Journal of Research and Development, 17(6):525–532, 1973.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman.
Concurrency Control and Recovery in Database Systems.
Addison-Wesley, 1987.

[14] L. Bocchi, C. Laneve, and G. Zavattaro.
A Calculus for Long-Running Transactions.
In *6th IFIP WG 6.1 International Conference Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 2884 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2003.

[15] B. Boothe.
Efficient algorithms for bidirectional debugging.
In *ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310. ACM, 2000.

[16] M. Boreale and D. Sangiorgi.
A Fully Abstract Semantics for Causality in the π-Calculus.
Acta Informatica, 35(5):353–400, 1998.

[17] G. Boudol.
Towards a Lambda-Calculus for Concurrent and Communicating Systems.
In *TAPSOFT, Vol.1*, pages 149–161, 1989.

[18] R. Bruni, H. C. Melgratti, and U. Montanari.

Flat Committed Join in Join.
Electr. Notes Theor. Comput. Sci., 104:39–59, 2004.

[19] R. Bruni, H. C. Melgratti, and U. Montanari.
Nested Commits for Mobile Calculi: Extending Join.
In *3rd International Conference on Theoretical Computer Science (TCS)*, pages 563–576. Kluwer, 2004.

[20] R. Bruni, H. C. Melgratti, and U. Montanari.
Theoretical foundations for compensations in flow composition languages.
In *32nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 209–220. ACM, 2005.

[21] H. Buhrman, J. Tromp, and P. M. B. Vitányi.
Time and Space Bounds for Reversible Simulation.
In *28th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2076 of *Lecture Notes in Computer Science*, pages 1017–1027. Springer, 2001.

[22] L. Cardelli and C. Laneve.
Reversibility in Massive Concurrent Systems.
CoRR, abs/1108.3419, 2011.

[23] L. Cardelli and C. Laneve.
Reversible structures.
In *Computational Methods in Systems Biology, 9th International Conference, CMSB 2011*, pages 131–140, 2011.

[24] C. D. Carothers, K. S. Perumalla, and R. Fujimoto.
Efficient optimistic parallel simulations using reverse computation.
ACM Transactions on Modeling and Computer Simulation, 9(3):224–253, 1999.

[25] G. L. Cattani and P. Sewell.
Models for name-passing processes: interleaving and causal.
Inf. Comput., 190(2):136–178, 2004.

[26] T. Chothia and D. Duggan.
Abstractions for fault-tolerant global computing.
Theor. Comput. Sci., 322(3):567–613, 2004.

[27] S. Crafa, D. Varacca, and N. Yoshida.
Compositional Event Structure Semantics for the Internal Pi-Calculus.
In *18th International Conference Concurrency Theory (CONCUR)*, volume 4703 of *Lecture Notes in Computer Science*. Springer, 2007.

[28] K. Czarnecki, J. N. Foster, Z. Hu, R. L, and A. Sch.
Bidirectional Transformations : A Cross-Discipline Perspective GRACE Meeting Notes , State of the Art , and Outlook.
Theory and Practice of Model Transformations, 5563:260–283, 2009.

[29] V. Danos and J. Krivine.
Reversible Communicating Systems.
In *15th International Conference on Concurrency Theory (CONCUR)*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[30] V. Danos and J. Krivine.
Transactions in RCCS.
In *16th International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[31] V. Danos and J. Krivine.
Formal Molecular Biology Done in CCS-R.
Electr. Notes Theor. Comput. Sci., 180(3):31–49, 2007.

[32] V. Danos and L. Regnier.
Reversible, Irreversible and Optimal lambda-Machines.
Theor. Comput. Sci., 227(1-2):79–97, 1999.

[33] R. De Nicola, U. Montanari, and F. W. Vaandrager.
Back and Forth Bisimulations.
In *CONCUR '90, Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 1990.

[34] E. de Vries, V. Koutavas, and M. Hennessy.
Communicating Transactions.
In *21st International Conference on Concurrency Theory (CONCUR)*, volume 6269 of *Lecture Notes in Computer Science*, pages 569–583. Springer, 2010.

[35] E. de Vries, V. Koutavas, and M. Hennessy.
Liveness of Communicating Transactions (Extended Abstract).
In *8th Asian Symposium Programming Languages and Systems (APLAS)*, volume 6461 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2010.

[36] P. Degano and C. Priami.
Enhanced operational semantics.
ACM Comput. Surv., 33(2):135–176, 2001.

[37] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson.
A survey of rollback-recovery protocols in message-passing systems.
ACM Computing Surveys, 34(3), 2002.

[38] J. L. Eppinger, L. B. Mummert, and A. Z. Spector.
Camelot and Avalon: A Distributed Transaction Facility.
Morgan Kaufmann, 1991.

[39] S. I. Feldman and C. B. Brown.
Igor: A system for program debugging via reversible execution.
In *Workshop on Parallel and Distributed Debugging*, 1988.

[40] R. P. Feynman.
Feynman Lectures on Computation.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[41] J. Field and C. A. Varela.
Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments.
In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 195–208. ACM, 2005.

[42] C. Fournet and G. Gonthier.
The reflexive chemical abstract machine and the join-calculus.
In *23rd ACM Symp. on Princ. of Prog. Languages (POPL)*, pages 372–385, 1996.

[43] C. Fournet and G. Gonthier.
The join calculus : a language for distributed mobile programming.
Applied Semantics, (September 2000):1–66, 2000.

[44] E. Fredkin and T. Toffoli.
Conservative logic.
International Journal of Theoretical Physics, 21:219–253, 1982.

[45] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem.
Modeling long-running activities as nested sagas.
IEEE Data Eng. Bull., 14(1):14–18, 1991.

[46] H. Garcia-Molina and K. Salem.
Sagas.
SIGMOD Rec., 16:249–259, December 1987.

[47] D. Gorla.
Towards a unified approach to encodability and separation results for process calculi.

Inf. Comput., 208(9):1031–1053, 2010.

[48] J. Gray and A. Reuter.
Transaction Processing: Concepts and Techniques.
Morgan Kaufmann, 1992.

[49] C. Guidi, I. Lanese, F. Montesi, and G. Zavattaro.
On the interplay between fault handling and request-response service invocations.
In *8th International Conference on Application of Concurrency to System Design (ACSD 2008)*, pages 190–198, 2008.

[50] N. Haines, D. Kindred, J. G. Morrisett, S. Nettles, and J. M. Wing.
Composing First-Class Transactions.
ACM Trans. Program. Lang. Syst., 16(6):1719–1736, 1994.

[51] I. Hasuo.
Generic Forward and Backward Simulations.
In *17th International Conference on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 447–461. Springer, 2006.

[52] M. Herlihy, J. Eliot, and B. Moss.
Transactional Memory: Architectural Support for Lock-Free Data Structures.
In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[53] B. Jacobs and F. Piessens.
Failboxes: Provably Safe Exception Handling.
In *23rd European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *Lecture Notes in Computer Science*, pages 470–494. Springer, 2009.

[54] S. Jagannathan, J. Vitek, A. Welc, and A. L. Hosking.
A transactional object calculus.
Science Computer Programming, 57(2):164–186, 2005.

[55] O. Laadan and J. Nieh.
Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems.
In *USENIX Annual Technical Conference*, pages 1–14. USENIX, 2007.

[56] R. Landauer.
Irreversibility and heat generation in the computing process.
IBM J. Res. Dev., 5:183–191, July 1961.

[57] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani.
Controlling Reversibility in Higher-Order Pi.
In *CONCUR*, pages 297–311, 2011.

[58] I. Lanese, C. A. Mezzina, and J.-B. Stefani.
Reversing Higher-Order Pi.
In *CONCUR*, pages 478–493, 2010.

[59] I. Lanese and F. Montesi.
Error Handling: From Theory to Practice.
In *ISoLA (2)*, pages 66–81, 2010.

[60] I. Lanese, C. Vaz, and C. Ferreira.
On the Expressive Power of Primitives for Compensation Handling.
In *19th European Symposium on Programming (ESOP)*, volume 6012 of *Lecture Notes in Computer Science*, pages 366–386. Springer, 2010.

[61] C. Laneve and G. Zavattaro.
Foundations of Web Transactions.
In *Foundations of Software Science and Computational Structures, 8th International Conference (FOSSACS)*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.

[62] C. Laneve and G. Zavattaro.
web-pi at Work.
In *Trustworthy Global Computing, International Symposium, TGC*, pages 182–194, 2005.

[63] G. B. Leeman.
A Formal Approach to Undo Operations in Programming Languages.
ACM Trans. Program. Lang. Syst., 8(1):50–87, 1986.

[64] S. Lenglet, A. Schmitt, and J.-B. Stefani.
Normal Bisimulations in Calculi with Passivation.
In *FOSSACS*, pages 257–271, 2009.

[65] B. Liskov.
Distributed programming in Argus.
Commun. ACM, 31:300–312, March 1988.

[66] B. Liskov and R. Scheifler.
Guardians and Actions: Linguistic Support for Robust, Distributed Programs.
ACM Trans. Program. Lang. Syst., 5(3):381–404, 1983.

[67] M. Little.

Web services transactions: past, present and future.
In *XML Conference and Exposition*, 2003.

[68] N. A. Lynch and F. W. Vaandrager.
Forward and Backward Simulations: I. Untimed Systems.
Inf. Comput., 121(2):214–233, 1995.

[69] N. Margolus.
Physics-like models of computation.
Physica D: Nonlinear Phenomena, 10(1-2):81 – 95, 1984.

[70] R. Milner.
A Calculus of Communicating Systems, volume 92 of *Lecture Notes in Computer Science*.
Springer, 1980.

[71] R. Milner.
Calculi for Synchrony and Asynchrony.
Theor. Comput. Sci., 25:267–310, 1983.

[72] R. Milner.
Functions as Processes.
Mathematical Structures in Computer Science, 2(2):119–141, 1992.

[73] R. Milner and D. Sangiorgi.
Barbed Bisimulation.
In *Automata, Languages and Programming, 19th International Colloquium, ICALP92*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer, 1992.

[74] F. Montesi, C. Guidi, I. Lanese, and G. Zavattaro.
Dynamic Fault Handling Mechanisms for Service-Oriented Applications.
In *ECOWS*, pages 225–234, 2008.

[75] K. F. Moore and D. Grossman.
High-level small-step operational semantics for transactions.
SIGPLAN Not., 43:51–62, January 2008.

[76] J. E. B. Moss.
Open Nested Transactions: Semantics and Support (poster).
In *Workshop on Memory Performance Issues*, Feb 2006.

[77] M. R. Mousavi, M. A. Reniers, and J. F. Groote.
SOS formats and meta-theory: 20 years after.
Theor. Comput. Sci., 373(3):238–272, 2007.

[78] C. Papadimitriou.
The theory of database concurrency control.
Computer Science Press, Inc., New York, NY, USA, 1986.

[79] J. A. Pérez.
Higher-Order Concurrency: Expressiveness and Decidability Results.
PhD thesis, University of Bologna, 2010.

[80] K. Peters, J.-W. Schicke, and U. Nestmann.
Synchrony vs Causality in the Asynchronous Pi-Calculus.
In *EXPRESS*, pages 89–103, 2011.

[81] A. Phillips and L. Cardelli.
A programming language for composable DNA circuits.
Journal of The Royal Society Interface, 6(Suppl 4):S419–S436, Aug. 2009.

[82] I. C. C. Phillips and I. Ulidowski.
Reversing algebraic process calculi.
J. Log. Algebr. Program., 73(1-2):70–96, 2007.

[83] I. C. C. Phillips and I. Ulidowski.
A Logic with Reverse Modalities for History-preserving Bisimulations.
In *EXPRESS*, pages 104–118, 2011.

[84] M. F. Ringenburg and D. Grossman.
AtomCaml: first-class atomicity via rollback.
In *10th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM, 2005.

[85] P. V. Roy and S. Haridi.
Concepts, Techniques, and Models of Computer Programming.
MIT Press, 2004.

[86] D. Sangiorgi.
Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.
PhD thesis CST–99–93, University of Edinburgh, Dept. of Comp. Sci., 1992.

[87] D. Sangiorgi.
Bisimulation for Higher-Order Process Calculi.
Information and Computation, 131(2), 1996.

[88] D. Sangiorgi.
From lambda to pi; or, Rediscovering continuations.
Mathematical Structures in Computer Science, 9(4):367–401, 1999.

[89] D. Sangiorgi and D. Walker.
The π-calculus: A Theory of Mobile Processes.
Cambridge University Press, 2001.

[90] T. Saridakis.
Design Patterns for Log-Based Rollback Recovery.
In *In Proceedings of the 2nd Nordic Conference on Pattern Languages of Programs (VikingPLoP)*, pages 7–17, 2003.

[91] A. Schmitt and J.-B. Stefani.
The m-calculus: a higher-order distributed process calculus.
In *POPL*, pages 50–61, 2003.

[92] H. Schuldt, G. Alonso, C. Beeri, and H. Schek.
Atomicity and isolation for transactional processes.
ACM Trans. Database Systems, 27(1):63–116, 2002.

[93] L. Szilard.
On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings.
Behavioral Science, 9(4):301–310, 1964.

[94] T. Toffoli.
Computation and Construction Universality of Reversible Cellular Automata.
J. Comput. Syst. Sci., 15(2):213–231, 1977.

[95] K. S. Trivedi.
Probability and statistics with reliability, queuing and computer science applications.
John Wiley and Sons Ltd., Chichester, UK, 2nd edition edition, 2002.

[96] D. Varacca and N. Yoshida.
Typed event structures and the linear pi-calculus.
Theoretical Computer Science, 411(19):1949–1973, 2010.

[97] P. M. B. Vitányi.
Time, space, and energy in reversible computing.
In *2nd Conference on Computing Frontiers*, pages 435–444. ACM, 2005.

[98] T. Yokoyama, H. B. Axelsen, and R. Glück.
Principles of a reversible programming language.
In *5th Conference on Computing Frontiers*, pages 43–54. ACM, 2008.

[99] T. Yokoyama, H. B. Axelsen, and R. Glück.
Reversible Flowchart Languages and the Structured Reversible Program Theorem.

In *35th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 5126 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 2008.

[100] T. Yokoyama and R. Glück.
A reversible programming language and its invertible self-interpreter.
In *PEPM*, pages 144–153, 2007.

[101] L. Ziarek and S. Jagannathan.
Lightweight checkpointing for concurrent ML.
J. Funct. Program., 20(2):137–173, 2010.

Appendix A

Encoding Proofs

A.1 Normal Form Properties

We want to characterize the effect of normal form on a process inside a context, i.e. a term $\mathbb{C}[P]$. To this end we define normal form on contexts, which also computes the substitution applied to the hole $\bullet$. Since we are working with contexts with one hole only, we require that for all the higher-order applications $(X)P\ \mathbb{C}_1[\bullet]$ either P is linear or P is already in normal form. In the first case the bullet is not replicated, and single-hole contexts are enough. In the second case, no inductive call is required.

Definition A.1 (Context normal form) *We define the context normal form function* $\mathtt{nfc}(\mathbb{C}[\bullet])$ *as* $\mathtt{nfc}(\mathbb{C}[\bullet], \emptyset)$*, where the second parameter is used for computing the substitution applied to the bullet (or to the bullets). The result is also a pair* $(\mathbb{C}'[\bullet], \sigma)$*. The function* $\mathtt{nfc}(\mathbb{C}[\bullet], \sigma)$ *is defined as follows:*

$$\mathtt{nfc}(P \mid \mathbb{C}_1[\bullet], \sigma) = \mathtt{nf}(P) \mid \mathbb{C}'[\bullet], \sigma' \ \ \textit{if}\ \mathtt{nfc}(\mathbb{C}_1[\bullet], \sigma) = \mathbb{C}'[\bullet], \sigma'$$

$$\mathtt{nfc}(\nu a.\, \mathbb{C}_1[\bullet], \sigma) = \nu a.\, \mathbb{C}'[\bullet], \sigma' \ \ \textit{if}\ \mathtt{nfc}(\mathbb{C}_1[\bullet], \sigma) = \mathbb{C}'[\bullet], \sigma'$$

$$\mathtt{nfc}((X)\mathbb{C}_1[\bullet]\ P, \sigma) = \mathbb{C}'[\bullet], \sigma' \ \ \textit{if}\ \mathtt{nfc}(\mathbb{C}_1\{^P/_X\}[\bullet], \sigma \cdot \{^P/_X\}) = \mathbb{C}'[\bullet], \sigma'$$

$$\mathtt{nfc}((X)P\ \mathbb{C}_1[\bullet], \sigma) = \mathbb{C}'[\bullet], \sigma' \ \ \textit{if}\ \mathtt{nfc}(P\{^{\mathbb{C}_1[\bullet]}/_X\}, \sigma) = \mathbb{C}', \sigma' \quad \textit{and } P \textit{ is linear}$$

$$\mathtt{nfc}((X)P\ \mathbb{C}_1[\bullet], \sigma) = P\{^{\mathbb{C}_1[\bullet]}/_X\}, \sigma \cdot \{^{\mathbb{C}_1[\bullet]}/_X\} \ \ \textit{if } P \textit{ is in normal form}$$

$$\mathtt{nfc}((h)\mathbb{C}_1[\bullet]\ l, \sigma) = \mathbb{C}'[\bullet], \sigma' \ \ \textit{if}\ \mathtt{nfc}(\mathbb{C}_1\{^l/_h\}[\bullet], \sigma \cdot \{^l/_h\}) = \mathbb{C}'[\bullet], \sigma'$$

$$\mathtt{nfc}(a\langle \mathbb{C}_1[\bullet]\rangle, \sigma) = a\langle \mathbb{C}_1[\bullet]\rangle, \sigma$$

$$\mathtt{nfc}(a(X) \triangleright \mathbb{C}_1[\bullet], \sigma) = a(X) \triangleright \mathbb{C}_1[\bullet], \sigma$$

$$\mathtt{nfc}(\bullet, \sigma) = \bullet, \sigma$$

Lemma A.1 *If* $\mathtt{nfc}(\mathbb{C}[\bullet], \emptyset) = \mathbb{C}_1, \sigma$ *then* $\mathtt{nfc}(\mathbb{C}[\bullet], \sigma') = \mathbb{C}_1, \sigma' \cdot \sigma$

Proof: By simply looking to its definition. □

We now show that the notions of normal form for processes and for contexts are compatible.

Lemma A.2 $\mathtt{nf}(\mathbb{C}[P]) = \mathbb{C}'[\mathtt{nf}(P\sigma)]$ *with* $\mathtt{nfc}(\mathbb{C}[\bullet]) = \mathbb{C}'[\bullet], \sigma$*.*

Proof: By structural induction on $\mathbb{C}[\bullet]$. We have the following possibilities.

- $\mathbb{C}[\bullet] = \bullet$) we banally have that $\mathtt{nfc}(\bullet, \emptyset) = \bullet, \emptyset$ and $\mathtt{nf}(P\emptyset) = \mathtt{nf}(P)$.

- $\mathbb{C}[\bullet] = \nu u.\, \mathbb{C}_1[\bullet]$) we have that $\mathtt{nf}(\nu u.\, \mathbb{C}_1[P]) = \nu u.\, \mathtt{nf}(\mathbb{C}_1[P])$. By inductive hypothesis we have that $\mathtt{nf}(\mathbb{C}_1[P]) = \mathbb{C}'_1[\mathtt{nf}(P\sigma)]$ with $\mathtt{nfc}(\mathbb{C}_1[\bullet], \emptyset) = \mathbb{C}'_1[\bullet], \sigma$, and since $\mathtt{nfc}(\nu u.\, \mathbb{C}_1[\bullet], \emptyset) = \nu u.\, \mathtt{nfc}(\mathbb{C}_1[\bullet], \emptyset)$, we can conclude.

- $\mathbb{C}[\bullet] = Q \mid \mathbb{C}_1[\bullet]$) we have that $\mathtt{nf}(Q \mid \mathbb{C}_1[P]) = \mathtt{nf}(Q) \mid \mathtt{nf}(\mathbb{C}_1[P])$. By inductive hypothesis we have that $\mathtt{nf}(\mathbb{C}_1[P]) = \mathbb{C}'_1[\mathtt{nf}(P\sigma)]$ with $\mathtt{nfc}(\mathbb{C}_1[P], \emptyset) = \mathbb{C}'_1, \sigma$, and we can conclude by noting that $\mathtt{nfc}(Q \mid \mathbb{C}_1[\bullet], \emptyset) = \mathtt{nf}(Q) \mid \mathtt{nfc}(\mathbb{C}_1[\bullet], \emptyset)$.

- $\mathbb{C}[\bullet] = a\langle \mathbb{C}_1[\bullet] \rangle$ or $\mathbb{C}[\bullet] = a(X) \triangleright \mathbb{C}_1[\bullet]$) Simpler than the above cases, since there is no recursive call.

- $\mathbb{C}[\bullet] = (X)\mathbb{C}_1[\bullet]\ Q$) we have to show that $\mathtt{nf}((X)\mathbb{C}_1[P]\ Q) = \mathbb{C}'[\mathtt{nf}(P\sigma)]$ with $\mathtt{nfc}((X)\mathbb{C}_1[\bullet]\ Q, \emptyset) = \mathbb{C}', \sigma$. By definition, $\mathtt{nfc}((X)\mathbb{C}_1[\bullet]\ Q, \emptyset) = \mathtt{nfc}(\mathbb{C}_1[\bullet]\{{}^{Q}/_{X}\}, \{{}^{Q}/_{X}\})$. We have $\mathtt{nf}(\mathbb{C}_1[P]\{{}^{Q}/_{X}\}) = \mathtt{nf}(\mathbb{C}_1\{{}^{Q}/_{X}\}[P\{{}^{Q}/_{X}\}]) = \mathbb{C}''[\mathtt{nf}(P\{{}^{Q}/_{X}\}\sigma')]$ where $\mathtt{nfc}(\mathbb{C}_1\{{}^{Q}/_{X}\}\ [\bullet], \emptyset) = \mathbb{C}'', \sigma'$ and by using Lemma A.1 $\mathtt{nfc}(\mathbb{C}_1\{{}^{Q}/_{X}\}[\bullet], \{{}^{Q}/_{X}\}) = \mathbb{C}'', \{{}^{Q}/_{X}\} \cdot \sigma'$. We have that $\mathbb{C}'', \sigma'\{{}^{Q}/_{X}\} = \mathbb{C}', \sigma$. So $\mathtt{nf}((X)\mathbb{C}_1[P]\ Q) = \mathbb{C}'[\mathtt{nf}(P\sigma)]$ with $\mathtt{nfc}((X)\mathbb{C}_1[\bullet]\ Q, \emptyset) = \mathbb{C}', \sigma$, as desired.

- $\mathbb{C}[\bullet] = \mathtt{nf}((h)\mathbb{C}_1[\bullet]\ v)$) we can apply the same reasoning of the above case.

- $\mathbb{C}[\bullet] = \mathtt{nf}((X)Q\ \mathbb{C}_1[\bullet])$ with Q linear) we have that $\mathtt{nf}((X)Q\ \mathbb{C}_1[P]) = \mathtt{nf}(Q\{{}^{\mathbb{C}_1[P]}/_{X}\})$, but since Q is linear we can write $Q\{{}^{\mathbb{C}_1[P]}/_{X}\} = \mathbb{C}_2[P]$, hence $\mathtt{nf}((X)Q\ \mathbb{C}_1[P]) = \mathtt{nf}(Q\{{}^{\mathbb{C}_1[P]}/_{X}\}) = \mathtt{nf}(\mathbb{C}_2[P])$. By inductive hypothesis we have that $\mathtt{nf}(\mathbb{C}_2[P]) = \mathbb{C}'_2[P\sigma]$ with $\mathtt{nfc}(\mathbb{C}_2[\bullet], \emptyset) = \mathbb{C}'_2, \sigma$, as desired.

- $\mathbb{C}[\bullet] = \mathtt{nf}((X)Q\ \mathbb{C}_1[\bullet])$ with Q in normal form) we have that $\mathtt{nf}((X)Q\ \mathbb{C}_1[P]) = \mathtt{nf}(Q\{{}^{\mathbb{C}_1[P]}/_{X}\}) = Q\{{}^{\mathbb{C}_1[P]}/_{X}\}$ and $\mathtt{nfc}((X)Q\ \mathbb{C}_1[P], \emptyset) = \mathbb{C}'_1[\bullet], \sigma$ with $\mathtt{nfc}(Q\{{}^{\mathbb{C}_1[\bullet]}/_{X}\}, \emptyset) = Q\{{}^{\mathbb{C}_1[\bullet]}/_{X}\}, \emptyset$ and $\mathbb{C}'_1[\bullet] = Q\{{}^{\mathbb{C}_1[\bullet]}/$

$_X\}$, and we can conclude by stating that $\mathbb{C}'_1[P\emptyset] = Q\{^{\mathbb{C}_1[P]}/_X\}$, as desired. Let us note that if Q is in normal form then $\mathtt{nf}(Q\{^P/_X\}) = Q\{^P/_X\}$, since the substitution will take place in non active contexts.

□

Lemma 5.4. If $\mathtt{nf}(T) \equiv_{Ex} \mathtt{nf}(T')$ then $\mathtt{nf}(\mathtt{addG}(T)) \equiv_{Ex} \mathtt{nf}(\mathtt{addG}(T'))$

Proof: By definition of $\mathtt{addG}$ we have that $T \equiv \mathbb{E}[0]$ and $T' \equiv \mathbb{E}'[0]$, with $\mathtt{addG}(T) \equiv \mathbb{E}[P]$ and $\mathtt{addG}(T') \equiv \mathbb{E}'[P]$. By using Lemma A.2 we have that $\mathtt{nf}(\mathbb{E}[P]) = \mathbb{E}_1[\mathtt{nf}(P\sigma)]$ and $\mathtt{nf}(\mathbb{E}'[P]) = \mathbb{E}_2[\mathtt{nf}(P\sigma')]$. Since all the processes added by the function $\mathtt{addG}$ are closed we have that $P\sigma = P\sigma' = P$ and $\mathtt{nf}(\mathbb{E}[P]) = \mathbb{E}_1[\mathtt{nf}(P)]$ and $\mathtt{nf}(\mathbb{E}'[\mathtt{nf}(P)]) = \mathbb{E}_2[\mathtt{nf}(P)]$. By hypothesis we have that $\mathtt{nf}(\mathbb{E}[0]) \equiv_{Ex} \mathtt{nf}(\mathbb{E}'[0])$, we have that $\mathbb{E}_1[\mathtt{nf}(0\sigma)] = \mathbb{E}_1[0] \equiv_{Ex} \mathbb{E}_2[0] = \mathbb{E}_2[\mathtt{nf}(0\sigma')]$, and we can conclude by saying that also $\mathbb{E}_1[\mathtt{nf}(P)] \equiv_{Ex} \mathbb{E}_2[\mathtt{nf}(P)]$, that is $\mathtt{nf}(\mathtt{addG}(T)) \equiv_{Ex} \mathtt{nf}(addG(T'))$, as desired. □

Lemma 5.5. If $\mathtt{nf}(T_1) \equiv_{Ex} \mathtt{nf}(T_2)$ and $\mathtt{nf}(T_1) \to T'_1$ then $\mathtt{nf}(T_2) \Rightarrow \mathtt{nf}(T'_2)$ with $\mathtt{nf}(T'_1) \equiv_{Ex} \mathtt{nf}(T'_2)$.

Proof: By case analysis on the used axiom $M \equiv_{Ex} N$ and on the structure of $\mathtt{nf}(T_1)$. Since we are dealing with normal form then the only $\to$ that can be applied are communications and hence we can express $\mathtt{nf}(T_1)$ as an active context $\mathbb{E}[a\langle R\rangle \mid a(X) \triangleright S]$. We will just consider few interesting cases, the remaining ones are similar.

$P \mid Q \equiv_{Ex} Q \mid P$**.** There are four places in which the axiom could be applied: in the context, in the message content, in the trigger continuation or in the context hole directly. Let us suppose that it is applied in the message content, we have that $\mathtt{nf}(T_1)$ can be written as $\mathbb{E}[a\langle\mathbb{C}[P \mid Q]\rangle \mid a(X) \triangleright S]$ and $\mathtt{nf}(T_2)$ as $\mathbb{E}[a\langle\mathbb{C}[Q \mid P]\rangle \mid a(X) \triangleright S]$. We have that $\mathtt{nf}(T_1) \to \mathbb{E}[S\{^{\mathbb{C}[P|Q]}/_X\}]$ and $\mathtt{nf}(T_2) \to \mathbb{E}[S\{^{\mathbb{C}[Q|P]}/_X\}]$, with $\mathbb{E}[S\{^{\mathbb{C}[P|Q]}/_X\}] \equiv_{Ex} \mathbb{E}[S\{^{\mathbb{C}[Q|P]}/_X\}]$, and we can conclude by applying Lemma 5.3.

$(\nu u.\, P) \mid Q \equiv_{Ex} \nu u.\, (P \mid Q)$**.** Let us consider the case in which the axiom is applied on the message content. We have that $\mathbb{E}[a\langle\mathbb{C}[(\nu u.\, P) \mid Q]\rangle \mid a(X) \triangleright S] \to \mathbb{E}[S\{^{\mathbb{C}[(\nu u.\, P)|Q]}/_X\}]$ and on the other side we have $\mathbb{E}[a\langle\mathbb{C}[\nu u.\, (P \mid Q)]\rangle \mid a(X) \triangleright S] \to \mathbb{E}[S\{^{\mathbb{C}[\nu u.\, (P|Q)]}/_X\}]$, and we can conclude by applying Lemma 5.3.

Ax.C. We have that $\mathtt{nf}(T_1) = \mathbb{E}[\mathtt{nf}(\mathtt{KillP}\ l\ h\ k)] = \mathbb{E}[l(Z)|h(W) \triangleright k\langle (h)\mathtt{Par}\ Z\ W\ l\rangle]$. Now if the communication is performed by the context $\mathbb{E}[\bullet]$ then the thesis banally follows. If the communication is done by the axiom itself, we have that in the context there are two messages of the form $l\langle P\rangle$ and $h\langle Q\rangle$, hence $\mathbb{E}[\mathtt{nf}(\mathtt{KillP}\ l\ h\ k)] \equiv \mathbb{E}'[l\langle P\rangle \mid h\langle Q\rangle \mid l(Z)|h(W) \triangleright k\langle (h)\mathtt{Par}\ Z\ W\ l\rangle] \rightarrow \mathbb{E}'[k\langle (h)\mathtt{Par}\ P\ Q\ l\rangle]$ and by expanding the definition of $\mathtt{Par}$ we have that $\mathtt{nf}(T_1) \rightarrow \mathbb{E}'[k\langle (h)(\nu l_1, l_2.\ (P\ l_1) \mid (Q\ l_2) \mid \mathtt{KillP}\ l_1\ l_2\ l)\rangle] \equiv_{Ex} \mathbb{E}'[k\langle (h)(\nu l_1, l_2.\ (Q\ l_1) \mid (P\ l_2) \mid \mathtt{KillP}\ l_2\ l_1\ l)\rangle]$, with $\mathtt{nf}(T_2) \rightarrow \mathbb{E}'[k\langle (h)(\nu l_1, l_2.\ (Q\ l_1) \mid (P\ l_2) \mid \mathtt{KillP}\ l_2\ l_1\ l)\rangle]$, as desired.

Ax.P. We have $\mathtt{nf}(T_1) = \mathbb{E}[l_1\langle \llparenthesis P \rrparenthesis\rangle \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)]$. The context can interact with the hole by reading messages on l_1 or l_2. By Lemma A.13 the only processes that can read from an l channel are those generated by $\mathtt{Rew}\ l$ or by the killer of the father of a process. The second case will be considered later on. In the first case we have that $\mathtt{nf}(T_1) \equiv \mathbb{E}[l_1(Z) \triangleright Z\ l_1 \mid l_1\langle \llparenthesis P \rrparenthesis\rangle \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)] \rightarrow \mathbb{E}[\llparenthesis P \rrparenthesis l_1 \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)]$. By using Lemma A.14 and the axiom $Ex.Unfold$ we have that $\llparenthesis P \rrparenthesis l_1 \equiv_{Ex} \mathtt{Rew}\ l_1 \mid l_1\langle \llparenthesis P \rrparenthesis\rangle$, and then $\mathbb{E}[\llparenthesis P \rrparenthesis l_1 \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)] \equiv_{Ex} \mathbb{E}[\mathtt{Rew}\ l_1 \mid l_1\langle \llparenthesis P \rrparenthesis\rangle \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)]$ So $\mathbb{E}[\mathtt{Rew}\ l_1 \mid l_1\langle \llparenthesis P \rrparenthesis\rangle \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid \mathtt{nf}(\mathtt{KillP}\ l_1\ l_2\ l)] \equiv_{Ex} \mathtt{nf}(\mathbb{E}[\mathtt{Rew}\ l_1 \mid l\langle (h)\mathtt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h\rangle \mid \mathtt{Rew}\ l])$, as desired.

If the hole evolves by itself we have that $\mathtt{nf}(T_1) = \mathbb{E}[l_1\langle \llparenthesis P \rrparenthesis\rangle \mid l_2\langle \llparenthesis Q \rrparenthesis\rangle \mid (l_1(Z)|l_2(W) \triangleright (l)\mathtt{Par}\ Z\ W\ h \mid \mathtt{Rew}\ l)] \rightarrow \mathbb{E}[(l)\mathtt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h \mid \mathtt{Rew}\ l]$ and the thesis banally follows. The case in which reductions of the right part of the axiom are considered are trivial since we have that the left part with a communication reduces to the right one.

Ax.A. We have that $\mathtt{nf}(T_1) = \mathbb{E}[\nu l'.\ l_1(Z)|l_2(W) \triangleright l'\langle (h)\mathtt{Par}\ Z\ W\ h \mid \mathtt{Rew}\ l'\rangle \mid l'(Z)|l_3(W) \triangleright l\langle (h)\mathtt{Par}\ Z\ W\ h \mid \mathtt{Rew}\ l\rangle]$. The only way for the hole to perform a communication is the presence of two messages on l_1, l_2 in the context. If so, we have that $\mathtt{nf}(T_1) \equiv \mathbb{E}'[\nu l'.\ l_1\langle P\rangle \mid l_2\langle Q\rangle \mid (l_1(Z)|l_2(W) \triangleright l'\langle (h)\mathtt{Par}\ Z\ W\ h\rangle \mid \mathtt{Rew}\ l') \mid \mathtt{nf}(\mathtt{KillP}\ l'\ l_3\ l)] \rightarrow \mathbb{E}[l'\langle (h)\mathtt{Par}\ P\ Q\ h\rangle \mid \mathtt{Rew}\ l' \mid \mathtt{nf}(\mathtt{KillP}\ l'\ l_3\ l)] \equiv_{Ex} \mathbb{E}[\nu l'.\ l_1\langle P\rangle \mid l_2\langle Q\rangle \mid (l_1(Z)|l_2(W) \triangleright l'\langle (h)\mathtt{Par}\ Z\ W\ h\rangle \mid \mathtt{Rew}\ l') \mid \mathtt{nf}(\mathtt{KillP}\ l'\ l_3\ l)]$, as desired.

Ax.Unfold. Since, $P \equiv \nu\tilde{u}.\ Q$, we have that $\mathtt{nf}(\llparenthesis P \rrparenthesis l) \equiv \mathtt{nf}(\llparenthesis \nu\tilde{u}.\ Q \rrparenthesis l) = \nu\tilde{u}.\ \mathtt{nf}(\llparenthesis Q \rrparenthesis l)$, hence $\mathtt{nf}(T_1) = \mathbb{E}[\nu\tilde{u}.\ \mathtt{nf}(\llparenthesis Q \rrparenthesis l)]$. Now if $\mathbb{E}[\nu\tilde{u}.\ \mathtt{nf}(\llparenthesis Q \rrparenthesis l)] \rightarrow R$ then also $\mathtt{nf}(T_2) = \mathbb{E}[\nu\tilde{u}.\ l\langle \llparenthesis Q \rrparenthesis\rangle \mid l(Z) \triangleright Z\ l] \rightarrow \mathbb{E}[\nu\tilde{u}.\ \llparenthesis Q \rrparenthesis l] \rightarrow^* \mathbb{E}[\nu\tilde{u}.\ \mathtt{nf}(\llparenthesis Q \rrparenthesis l)] \rightarrow R$, as desired. If the right part of the axiom moves, we have that a communication can be done by the context or by the hole itself. In the first case even the left part can mimic the step since the context is the same. In the second case we have that the hole can do just one communication whose effect is to reduces to left part, that is $\mathbb{E}[\nu\tilde{u}.\, l\langle(\!|Q|\!)\rangle \mid l(Z) \triangleright Z\ l] \to \mathbb{E}[\nu\tilde{u}.\, (\!|Q|\!)l]$ and $\mathtt{nf}(\mathbb{E}[\nu\tilde{u}.\, (\!|Q|\!)l]) = \mathtt{nf}(T_1)$, as desired.

Ax.Adm. By simply noting that the right part of the axiom can only perform a communication, reducing to the right part of the encoding.

□

Lemma A.3 *For all names k and $\rho\pi$ processes P, Q, if $P \equiv Q$ then* $\mathtt{nf}((\!|P|\!)k) \equiv_{Ex} \mathtt{nf}((\!|Q|\!)k)$

Proof: By induction on the derivation of $P \equiv Q$. The only interesting case is the base one. We have a case analysis on the applied axiom.

$P \mid 0 \equiv P$. We have that $\mathtt{nf}((\!|P \mid 0|\!)k) = \mathtt{nf}((\!|P|\!)k)$ (by definition of $(\!|P \mid 0|\!)$ in Figure 5.3), and we can conclude.

$P \mid Q \equiv Q \mid P$. If one of P or Q is equivalent to 0, the thesis banally follows. In the other case, we have that $\mathtt{nf}((\!|P \mid Q|\!)k) = \nu l, h.\, \mathtt{nf}((\!|P|\!)l) \mid \mathtt{nf}((\!|Q|\!)h) \mid \mathtt{nf}(\mathtt{KillP}\ l\ h\ k) \equiv_{Ex} \nu l, h.\, \mathtt{nf}((\!|P|\!)l) \mid \mathtt{nf}((\!|P|\!)h) \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k) = \mathtt{nf}((\!|Q \mid P|\!)k)$, as desired.

$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$. If at least one of P, Q or R are equivalent to 0 the thesis banally follows. In the other case we have that $\mathtt{nf}((\!|P \mid (Q \mid R)|\!)k) = \nu h, l, h', l'.\, \mathtt{nf}((\!|P|\!)h) \mid \mathtt{nf}((\!|Q|\!)h') \mid \mathtt{nf}((\!|R|\!)l') \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k) \mid \mathtt{nf}(\mathtt{KillP}\ h'\ l'\ l)$ and by using α-conversion we have that $\nu h, l, h', l'.\, \mathtt{nf}((\!|P|\!)h) \mid \mathtt{nf}((\!|Q|\!)h') \mid \mathtt{nf}((\!|R|\!)l') \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k) \mid \mathtt{nf}(\mathtt{KillP}\ h'\ l'\ l) \equiv_{Ex} \nu h, l, h', l'.\, \mathtt{nf}((\!|P|\!)h') \mid \mathtt{nf}((\!|Q|\!)l') \mid \mathtt{nf}((\!|R|\!)l) \mid \mathtt{nf}(\mathtt{KillP}\ h'\ h\ k) \mid \mathtt{nf}(\mathtt{KillP}\ l'\ l\ h) \equiv_{Ex} \nu h, l, h', l'.\, \mathtt{nf}((\!|P|\!)h') \mid \mathtt{nf}((\!|Q|\!)l') \mid \mathtt{nf}((\!|R|\!)l) \mid \mathtt{nf}(\mathtt{KillP}\ h'\ l'\ h) \mid \mathtt{nf}(\mathtt{KillP}\ l\ h\ k) \equiv_{Ex} \mathtt{nf}((\!|(P \mid Q) \mid R|\!)k)$, as desired.

$\nu u.\, 0 \equiv 0$. We have that $\mathtt{nf}((\!|\nu u.\, 0|\!)k) = \nu u.\, k\langle\mathtt{Nil}\rangle \mid \mathtt{nf}(\mathtt{Rew}\ k) \equiv_{Ex} k\langle\mathtt{Nil}\rangle \mid \mathtt{nf}(\mathtt{Rew}\ k) = \mathtt{nf}((\!|0|\!)k)$, as desired.

$(\nu u.\, P) \mid Q \equiv \nu u.\, (P \mid Q)$. If P or Q are equivalent to 0 then the thesis banally follows. In the other case, $\mathtt{nf}((\!|(\nu u.\, P) \mid Q|\!)k) = \nu l, h.\, \mathtt{nf}((\!|\nu u.\, P|\!)h) \mid \mathtt{nf}((\!|Q|\!)l) \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k) = \nu l, h.\, (\nu u.\, (\!|P|\!)h) \mid \mathtt{nf}((\!|Q|\!)l) \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k) \equiv_{Ex} \nu u.\, ((\!|P|\!)h \mid \mathtt{nf}((\!|Q|\!)l) \mid \mathtt{nf}(\mathtt{KillP}\ h\ l\ k)) = \mathtt{nf}((\!|\nu u.\, (P \mid Q)|\!)k)$, as desired.

$\nu a.\, \nu b.\, P \equiv \nu b.\, \nu a.\, P$. We have that $\mathtt{nf}(\llparenthesis \nu a.\, \nu b.\, P \rrparenthesis k) = \nu a.\, \nu b.\, \mathtt{nf}(\llparenthesis P \rrparenthesis k) \equiv_{Ex} \nu b.\, \nu a.\, \mathtt{nf}(\llparenthesis P \rrparenthesis k) = \mathtt{nf}(\llparenthesis \nu b.\, \nu a.\, P \rrparenthesis k)$, as desired.

α-conversion. The thesis banally follows.

□

Lemma 5.6. Let M, N be closed consistent configurations. Then $M \equiv N$ implies $\mathtt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \mathtt{nf}(\llparenthesis N \rrparenthesis)$.

Proof: By induction on the derivation of $M \equiv N$. The only interesting case is the base one. We have a case analysis on the applied axiom.

$M \mid 0 \equiv M$. By definition $\mathtt{nf}(\llparenthesis M \mid 0 \rrparenthesis) = \mathtt{nf}(\llparenthesis M \rrparenthesis) \mid \mathtt{nf}(\llparenthesis 0 \rrparenthesis) = \mathtt{nf}(\llparenthesis M \rrparenthesis) \mid \mathtt{nf}(0) \equiv_{Ex} \mathtt{nf}(\llparenthesis M \rrparenthesis)$.

$M \mid N \equiv N \mid M$. By definition $\mathtt{nf}(\llparenthesis M \mid N \rrparenthesis) = \mathtt{nf}(\llparenthesis M \rrparenthesis) \mid \mathtt{nf}(\llparenthesis N \rrparenthesis) \equiv_{Ex} \mathtt{nf}(\llparenthesis N \rrparenthesis) \mid \mathtt{nf}(trM) = \mathtt{nf}(\llparenthesis N \mid M \rrparenthesis)$.

$N_1 \mid (N_2 \mid N_3) \equiv (N_1 \mid N_2) \mid N_3$. By definition $\mathtt{nf}(\llparenthesis N_1 \mid (N_2 \mid N_3) \rrparenthesis) = \mathtt{nf}(\llparenthesis N_1 \rrparenthesis) \mid \mathtt{nf}(\llparenthesis N_2 \mid N_3 \rrparenthesis) = \mathtt{nf}(\llparenthesis N_1 \rrparenthesis) \mid \mathtt{nf}(\llparenthesis N_2 \rrparenthesis) \mid \mathtt{nf}(\llparenthesis N_3 \rrparenthesis) = \mathtt{nf}(\llparenthesis N_1 \mid N_2 \rrparenthesis) \mid \mathtt{nf}(\llparenthesis N_3 \rrparenthesis) = \mathtt{nf}(\llparenthesis (N_1 \mid N_2) \mid N_3 \rrparenthesis)$.

$\nu u.\, 0 \equiv 0$. By definition $\mathtt{nf}(\llparenthesis \nu u.\, 0 \rrparenthesis) = \nu u.\, \mathtt{nf}(\llparenthesis 0 \rrparenthesis) = \nu u.\, 0 \equiv_{Ex} 0 = \mathtt{nf}(\llparenthesis 0 \rrparenthesis)$.

$\nu u.\, \nu v.\, M \equiv \nu v.\, \nu u.\, M$. By definition $\mathtt{nf}(\llparenthesis \nu u.\, \nu v.\, M \rrparenthesis) = \nu u.\, \mathtt{nf}(\llparenthesis \nu v.\, M \rrparenthesis) = \nu u.\, \nu v.\, \mathtt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \nu v.\, \nu u.\, \mathtt{nf}(\llparenthesis M \rrparenthesis) = \nu v.\, \mathtt{nf}(\llparenthesis \nu u.\, M \rrparenthesis) = \mathtt{nf}(\llparenthesis \nu v.\, \nu u.\, M \rrparenthesis)$.

$(\nu u.\, M) \mid N \equiv \nu u.\, (M \mid N)$. By definition $\llparenthesis (\nu u.\, M) \mid N \rrparenthesis = \llparenthesis (\nu u.\, M) \rrparenthesis \mid \mathtt{nf}(\llparenthesis N \rrparenthesis) = (\nu u.\, \llparenthesis M \rrparenthesis) \mid \llparenthesis N \rrparenthesis \equiv_{Ex} \nu u.\, (\llparenthesis M \rrparenthesis \mid \llparenthesis N \rrparenthesis) = \nu u.\, (\llparenthesis M \mid N \rrparenthesis) \llparenthesis \nu u.\, (M \mid N) \rrparenthesis$.

$\kappa : \nu a.\, P \equiv \nu a.\, \kappa : P$. Here we have to distinguish two cases depending on the form of κ. If $\kappa = k$ then by definition $\mathtt{nf}(\llparenthesis k : \nu a.\, P \rrparenthesis) = \mathtt{nf}(\llparenthesis \nu a.\, P \rrparenthesis k) = \nu a.\, \mathtt{nf}(\llparenthesis P \rrparenthesis k) = \nu a.\, \mathtt{nf}(\llparenthesis k : P \rrparenthesis) = \mathtt{nf}(\llparenthesis \nu a.\, k : P \rrparenthesis)$. If $\kappa = \langle h_i, \tilde{h} \rangle \cdot k$ then by definition $\mathtt{nf}(\llparenthesis \langle h_i, \tilde{h} \rangle \cdot k : \nu a.\, P \rrparenthesis) = \mathtt{nf}(\llparenthesis \nu a.\, P \rrparenthesis h_i) \mid \mathtt{nf}(\mathtt{Kill}_{\langle h_i, \tilde{h} \rangle \cdot k}) = \nu a.\, \mathtt{nf}(\llparenthesis P \rrparenthesis h_i) \mid \mathtt{nf}(\mathtt{Kill}_{\langle h_i, \tilde{h} \rangle \cdot k}) = (\nu a.\, \mathtt{nf}(\llparenthesis P \rrparenthesis h_i) \mid \mathtt{Kill}_{\langle h_i, \tilde{h} \rangle \cdot k} \equiv_{Ex} \nu a.\, \mathtt{nf}(\llparenthesis P \rrparenthesis h_i) \mid \mathtt{nf}(\mathtt{Kill}_{\langle h_i, \tilde{h} \rangle \cdot k}) = \nu a.\, \mathtt{nf}(\llparenthesis \langle h_i, \tilde{h} \rangle \cdot k : P \rrparenthesis) = \mathtt{nf}(\llparenthesis \nu a.\, \langle h_i, \tilde{h} \rangle \cdot k : P \rrparenthesis)$.

$M =_\alpha N \Longrightarrow M \equiv N$. Since $\llparenthesis \_ \rrparenthesis$ and $\mathtt{nf}(\_)$ do not change bound variables we have that $\mathtt{nf}(\llparenthesis M \rrparenthesis) =_\alpha \mathtt{nf}(\llparenthesis N \rrparenthesis)$ and then $\mathtt{nf}(\llparenthesis M \rrparenthesis) \equiv_{Ex} \mathtt{nf}(\llparenthesis M \rrparenthesis)$.

$k : \prod_{i=1}^{n} \tau_i \equiv \nu\tilde{h}.\ \prod_{i=1}^{n}(\langle h_i, \tilde{h}\rangle \cdot k : \tau_i)$. By definition:

$$\begin{aligned}
&\texttt{nf}(\llparenthesis k : \prod_{i=1}^{n} \tau_i \rrparenthesis) = \texttt{nf}(\llparenthesis \prod_{i=1}^{n} \tau_i \rrparenthesis k) = \\
&\nu h_1, l_1.\, \texttt{nf}(\llparenthesis \tau_1 \rrparenthesis h_1) \mid \texttt{nf}(\llparenthesis \prod_{i=2}^{n} \tau_i \rrparenthesis l_1) \mid \texttt{nf}(\texttt{KillP}\ h_1\ l_1\ k) = \\
&\nu h_1, h_2, l_1, l_2.\, \texttt{nf}(\llparenthesis \tau_1 \rrparenthesis h_1) \mid \texttt{nf}(\llparenthesis \tau_2 \rrparenthesis h_2) \mid \texttt{nf}(\llparenthesis \prod_{i=2}^{n} \tau_i \rrparenthesis l_1) \mid \\
&\qquad\qquad \texttt{nf}(\texttt{KillP}\ h_1\ l_1\ k) \mid \texttt{nf}(\texttt{KillP}\ h_2\ l_2\ l_1) = \\
&\nu\tilde{h}, \tilde{l}.\ \prod_{i=1}^{n-1} \texttt{nf}(\llparenthesis \tau_i \rrparenthesis h_i) \mid \texttt{nf}(\llparenthesis \tau_n \rrparenthesis l_{n-1}) \mid \texttt{nf}(\texttt{KillP}\ h_1\ l_1\ k) \mid \\
&\qquad \prod_{i=2}^{n-1} \texttt{nf}(\texttt{KillP}\ h_i\ l_i\ l_{i-1})
\end{aligned}$$

Now by $\alpha-$converting the key of the last τ_n from l_{n-1} into h_n we obtain a term of the form

$$\begin{aligned}
&\nu\tilde{h}, \tilde{l}.\ \prod_{i=1}^{n} \texttt{nf}(\llparenthesis \tau_i \rrparenthesis h_i) \mid \texttt{nf}(\texttt{KillP}\ h_1\ l_1\ k) \mid \prod_{i=2}^{n-2} \texttt{nf}(\texttt{KillP}\ h_i\ l_i\ l_{i-1}) \mid \\
&\qquad \texttt{nf}(\texttt{KillP}\ h_{n-1}\ h_n\ l_{i-2}) = \\
&\texttt{nf}(\llparenthesis \nu\tilde{h}.\ \prod_{i=1}^{n} \langle h_i, \tilde{h}\rangle \cdot k : \tau_i \rrparenthesis)
\end{aligned}$$

Note that in this case we assumed that the $\mid$ is right associative, in order to unroll the parallel composition from $\prod_{i=1}^{n} \llparenthesis \tau_i \rrparenthesis$ to $\llparenthesis \tau_1 \rrparenthesis \mid \prod_{i=2}^{n} \llparenthesis \tau_i \rrparenthesis$.

□

A.2 Invariants

In this section we will prove some invariants of the encoding.

Lemma A.4 *For each $\rho\pi$ process P, $fn(\llparenthesis P \rrparenthesis) \cap \mathcal{K} = \emptyset$.*

Proof: By induction on the structure of P.

$P = 0$: by definition $\llparenthesis 0 \rrparenthesis = (l)\texttt{Nil} = (l)(l\langle \texttt{Nil} \rangle \mid \texttt{Rew}\ l) = P'$. We have that $fn(P') = \emptyset$ so the property is banally verified.

$P = a\langle Q\rangle$**:** by definition $\llparenthesis a\langle Q\rangle\rrparenthesis = (l)(\texttt{Msg}\ a\ \llparenthesis Q\rrparenthesis\ l) = (l)((a\ X\ l)(a\langle X,l\rangle \mid ((a\ l)(a(X,\backslash l) \triangleright l\langle (h)\texttt{Msg}\ a\ X\ h\rangle)\ a\ l) \mid ((l)(l(Z) \triangleright Z\ l)\ l)\ a\ \llparenthesis Q\rrparenthesis\ l)$. We have that $fn(\llparenthesis P\rrparenthesis) = (\{a\} \cup fn(\llparenthesis Q\rrparenthesis)) \setminus \{l\} = \{a\} \cup (fn(\llparenthesis Q\rrparenthesis) \setminus \{l\})$. By inductive hypothesis we have that $fn(\llparenthesis Q\rrparenthesis) \cap \mathcal{K} = \emptyset$, so also $(fn(\llparenthesis Q\rrparenthesis) \setminus \{l\}) \cap \mathcal{K} = \emptyset$. To correctly conclude we have to show that $(\{a\} \cup (fn(\llparenthesis Q\rrparenthesis)) \setminus \{l\}) \cap \mathcal{K} = \emptyset$, but since by definition $a \in \mathcal{N}$ and $\mathcal{N} \cap \mathcal{K} = \emptyset$ it follows that $fn(\llparenthesis P\rrparenthesis) \cap \mathcal{K} = \emptyset$.

$P = a(X) \triangleright Q$**:** by definition $\llparenthesis a(X) \triangleright Q\rrparenthesis = (l)(\texttt{Trig}\ ((X)(\llparenthesis Q\rrparenthesis\ X))\ a\ l) = (l)(\nu\texttt{t}.\,\overline{\texttt{t}} \mid (a(X,h)|\texttt{t}\triangleright\nu k.\,(((X)(\llparenthesis Q\rrparenthesis\ X))X)k \mid (\texttt{Mem}\ ((X)(\llparenthesis Q\rrparenthesis\ X))\ a\ X\ h\ k\ l) \mid (\texttt{KillT}((X)(\llparenthesis Q\rrparenthesis\ X))\ \texttt{t}\ l\ a))) = (l)(\nu\texttt{t}.\,\overline{\texttt{t}} \mid (a(X,h)|\texttt{t}\triangleright\nu k.\,(((X)(\llparenthesis Q\rrparenthesis\ X))X)k \mid k(Z)\triangleright(\texttt{Msg}\ a\ X\ h) \mid (\texttt{Trig}\ ((X)(\llparenthesis Q\rrparenthesis\ X))\ a\ l) \mid (\texttt{t}\triangleright l\langle (h)\texttt{Trig}\ ((X)(\llparenthesis Q\rrparenthesis\ X))\ a\ h\rangle)) = P'$. Now we have that $fn(P') = (\{a\} \cup fn(\llparenthesis Q\rrparenthesis)) \setminus \{l, k, t\texttt{t}\} = \{a\} \cup (fn(\llparenthesis Q\rrparenthesis) \setminus \{l, k, t\})$. By inductive hypothesis we have $fn(\llparenthesis Q\rrparenthesis) \cap \mathcal{K} = \emptyset$ so also $(fn(\llparenthesis Q\rrparenthesis) \setminus \{l, k, t\}) \cap \mathcal{K} = \emptyset$. By definition $a \in \mathcal{N}$ and $\mathcal{N} \cap \mathcal{K} = \emptyset$, hence it follows that $(\{a\} \cup (fn(\llparenthesis Q\rrparenthesis) \setminus \{l, k, \texttt{t}\})) \cap \mathcal{K} = \emptyset$ as desired.

$P = \nu a.\,Q$**:** by definition $\llparenthesis P\rrparenthesis = (l)\nu a.\,\llparenthesis Q\rrparenthesis l$. So $fn(\llparenthesis P\rrparenthesis) = fn(\llparenthesis Q\rrparenthesis) \setminus \{l, a\}$. By inductive hypothesis we have that $fn(\llparenthesis Q\rrparenthesis) \cap \mathcal{K} = \emptyset$, thus $(fn(\llparenthesis Q\rrparenthesis \setminus \{l, a\}) \cap \mathcal{K} = \emptyset$, as desired.

$P = X$**:** by definition $\llparenthesis X\rrparenthesis = X$. We have that $fn(\llparenthesis X\rrparenthesis) = \emptyset$ and the thesis banally follows.

$P = Q \mid R$**:** cases in which one of P or Q are equivalent to 0 banally holds. In the other case f by definition $\llparenthesis Q \mid R\rrparenthesis = (l)(\texttt{Par}\ \llparenthesis Q\rrparenthesis\ \llparenthesis R\rrparenthesis\ l) = (l)(\nu h, k.\,\llparenthesis Q\rrparenthesis h \mid \llparenthesis R\rrparenthesis k \mid (h(Z)|k(W) \triangleright l\langle (l)\texttt{Par}\ Z\ W\ l\rangle) \mid l(Z) \triangleright Zl))$. We have that $(fn(\llparenthesis P\rrparenthesis) = (\texttt{fn}(\llparenthesis Q\rrparenthesis) \cup fn(\llparenthesis R\rrparenthesis)) \setminus \{l, h, k\} = (\texttt{fn}(\llparenthesis Q\rrparenthesis) \setminus \{l, h, k\}) \cup (\texttt{fn}(\llparenthesis R\rrparenthesis) \setminus \{l, h, k\})$. By inductive hypothesis we have that $fn(\llparenthesis Q\rrparenthesis) \cap \mathcal{K} = \emptyset$ and $fn(\llparenthesis R\rrparenthesis) \cap \mathcal{K} = \emptyset$, so it follows that $(\texttt{fn}(\llparenthesis Q\rrparenthesis) \setminus \{l, h, k\}) \cap \mathcal{K} = \emptyset$ and $(\texttt{fn}(\llparenthesis R\rrparenthesis) \setminus \{l, h, k\}) \cap \mathcal{K} = \emptyset$, and also $((\texttt{fn}(\llparenthesis Q\rrparenthesis) \setminus \{l, h, k\}) \cup (\texttt{fn}(\llparenthesis R\rrparenthesis) \setminus \{l, h, k\})) \cap \mathcal{K} = \emptyset$, as desired.

□

Lemma A.5 *Be P an HOπ^+process. If $P \to Q$ then $\texttt{fn}(Q) \subseteq \texttt{fn}(P)$.*

Proof: By induction on $P \to Q$ and a case analysis on the last applied rule.

App: we have that $(\psi)F\ V \to F\theta$ with $\theta = \mathit{match}(V, \psi)$. By definition $\texttt{fn}((\psi)F\ V) = (\texttt{fn}(F) \setminus \mathit{dom}(\theta)) \cup \texttt{fn}(V)$ and $\texttt{fn}(F\theta) = (\texttt{fn}(F) \setminus \mathit{dom}(\theta)) \cup \texttt{fn}(\mathit{cod}(\theta))$, and we can conclude by noticing that $\texttt{fn}(\mathit{cod}(\theta)) \subseteq \texttt{fn}(V)$.

Red: we have that $a\langle F, v\rangle \mid a(X, h) \triangleright P$. By definition $\mathtt{fn}(a\langle F, v\rangle \mid a(X, h) \triangleright P) = \mathtt{fn}(a\langle F, v\rangle) \cup \mathtt{fn}(a(X, h) \triangleright P) = \{a, v\} \cup \mathtt{fn}(F) \cup (\mathtt{fn}(P) \setminus \{h\})$. By applying the reduction we have that $a\langle F, v\rangle \mid a(X, h) \triangleright P \to P\{^{v,F}/_{h,X}\}$ and $\mathtt{fn}(P\{^{v,F}/_{h,X}\}) = (\mathtt{fn}(P) \setminus \{h\}) \cup \{v\} \cup \mathtt{fn}(F)$, as desired.

Equiv: We have that $P \to Q$ with $P \equiv P_1$, $P_1 \to Q_1$ and $Q_1 \equiv Q$. By inductive hypothesis we have that $\mathtt{fn}(Q_1) \subseteq \mathtt{fn}(P_1)$ and the set of free names does not change under $\equiv$, we have that $\mathtt{fn}(Q) \subseteq \mathtt{fn}(P)$.

Ctx: If the context is the empty one we can directly conclude by applying the inductive hypothesis. If the context is a parallel, we have that $P \mid P_1 \to Q \mid P_1$, with $P \to Q$. By inductive hypothesis we have that $\mathtt{fn}(Q) \subseteq \mathtt{fn}(P)$ and we can conclude by saying that $\mathtt{fn}(P_1 \mid Q) \subseteq \mathtt{fn}(P \mid Q)$, as desired. If the context is a restriction, we have that $\nu u.\, P \to \nu u.\, Q$, with $P \to Q$. By inductive hypothesis we have that $\mathtt{fn}(Q) \subseteq \mathtt{fn}(P)$ and hence $\mathtt{fn}(Q) \setminus \{u\} \subseteq \mathtt{fn}(P) \setminus \{u\}$, as desired.

□

Lemma A.6 *Be P an HOπ^+process with $fn(P) \cap \mathcal{K} = \emptyset$. If $P \to Q$ then $fn(Q) \cap \mathcal{K} = \emptyset$*

Proof: By using Lemma A.5 we have that $P \to Q$ then $\mathtt{fn}(Q) \subseteq \mathtt{fn}(P)$, and we banally have that if $fn(P) \cap \mathcal{K} = \emptyset$ then also $fn(Q) \cap \mathcal{K} = \emptyset$. □

The following Lemma shows that all the keys generated by the encoding are restricted, that is all the messages carried on key channels will not be considered as observables, since all the key channels are restricted.

Lemma 5.7. If $(\!|\nu k.\, k : P|\!) \Rightarrow P'$ then $\mathtt{fn}(P') \cap \mathcal{K} = \emptyset$.

Proof: By induction on the number of reduction of $\Rightarrow$. We distinguish the base case and the inductive one. For the base case we have $P' = (\!|\nu k.\, k : P|\!) = \nu k.\, (\!|P|\!)k$. We have that $fn(P') = fn(\!|P|\!) \setminus \{k\}$ and by applying Lemma A.4 we obtain $fn((\!|P|\!)) \cap \mathcal{K} = \emptyset$ and hence $fn((\!|P'|\!)) \cap \mathcal{K} = \emptyset$ as desired. For the inductive case we have that $(\!|\nu k.\, k : P|\!) \Rightarrow P'' \to P'$ and we can use as inductive hypothesis $fn(P'') \cap \mathcal{K} = \emptyset$. Now we can conclude by applying Lemma A.6 and we are done. □

The following lemma states that all the messages on channels $a \in \mathcal{N}$ carries a couple, and that the first member of the couple is a translation of a process.

Lemma A.7 *For any $\rho\pi$ process P, if $\llparenthesis P \rrparenthesis k \Rightarrow \mathbb{E}[a\langle Q, h\rangle]$ with $a \in \mathcal{N}$ then $Q = \llparenthesis R \rrparenthesis$.*

Proof: By induction on the number of steps. The base case holds by a simple looking at the encoding of Figure 5.3. The inductive case banally holds. □

Definition A.2 *Let $Y = (X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle$. Then we denote P_l an HOπ^+process of one of the forms below:*

$$\begin{array}{ll}
\llparenthesis P \rrparenthesis l & l\langle \llparenthesis P \rrparenthesis \rangle \mid \mathtt{Rew}\, l \\
\mathtt{Msg}\, a\, \llparenthesis Q \rrparenthesis\, l & \mathtt{Trig}\, Y\, a\, l \\
\mathtt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ l & \mathtt{KillP}\, h\, k\, l \\
\nu c.\, (Y\ \llparenthesis P \rrparenthesis\ c) \mid (c(Z) \triangleright Z\, l) \mid (\mathtt{Mem}\, Y\ a\ \llparenthesis P \rrparenthesis\ l_1\ l\ l_2) & \nu a.\, \llparenthesis P \rrparenthesis l \\
\nu c.\, c\langle \llparenthesis P \rrparenthesis \rangle \mid (c(Z) \triangleright Z\, l) \mid (\mathtt{Mem}\, Y\ a\ \llparenthesis P \rrparenthesis\ l_1\ l\ l_2) & 0 \\
\mathtt{Mem}\, Y\ a\ \llparenthesis P \rrparenthesis\, h\, k\, l & \mathtt{Mem}\, Y\ a\ \llparenthesis P \rrparenthesis\, l\, k\, h
\end{array}$$

or a form obtained from the ones above via applications.

Lemma A.8 *For any $\rho\pi$ process P and any $l \in \mathtt{n}(\llparenthesis P \rrparenthesis) \cap \mathcal{K}$, then $\llparenthesis P \rrparenthesis = \mathbb{C}[(l)P_l]$ with $\mathbb{C}$ generated by $\mathbb{C} ::= \bullet \ \mid\ (l')\mathtt{Msg}\ a\ \mathbb{C}\ l' \ \mid\ (l')\mathtt{Trig}\ (X\ c)c\langle \mathbb{C} \rangle\ a\ l' \ \mid\ (l')\nu a.\, (\mathbb{C}\ l') \ \mid\ (l')\mathtt{Par}\ \mathbb{C}\ \llparenthesis Q \rrparenthesis\ l' \ \mid\ (l')\mathtt{Par}\ \llparenthesis P \rrparenthesis\ \mathbb{C}\ l'$.*

Proof: By structural induction on P.

$P = 0$, we have that $\llparenthesis 0 \rrparenthesis = \mathtt{Nil} = (l)(l\langle \mathtt{Nil} \rangle \mid \mathtt{Rew}\ l))$, as desired.

$P = a\langle Q \rangle$, we have that $\llparenthesis a\langle Q \rangle \rrparenthesis = (l)(\mathtt{Msg}\ a\ \llparenthesis Q \rrparenthesis\ l))$ and the thesis holds for the name l. By inductive hypothesis on $\llparenthesis Q \rrparenthesis$ for any name $l' \in \mathtt{n}(\llparenthesis Q \rrparenthesis \cap \mathcal{K})$ we have that $\llparenthesis Q \rrparenthesis = \mathbb{C}[(l')P_{l'}]$, thus $\llparenthesis a\langle Q \rangle \rrparenthesis = (l)(\mathtt{Msg}\ a\ \mathbb{C}[(l')P_{l'}]\ l)$ as desired.

$P = a(X) \triangleright Q$, we have that $\llparenthesis a(X) \triangleright Q \rrparenthesis = ((l)\mathtt{Trig}\ ((X\ c)c\langle \llparenthesis Q \rrparenthesis \rangle)\ \ a\, l))$, and the thesis holds for name l. For names in $\llparenthesis Q \rrparenthesis$ we can conclude by applying the inductive hypothesis.

$P = \nu a.\, P_1$, we have that $\llparenthesis \nu a.\, P_1 \rrparenthesis = ((l)\nu a.\, \llparenthesis P_1 \rrparenthesis l)$, and the thesis holds for name l. For names in $\llparenthesis P_1 \rrparenthesis$ we can conclude by applying the inductive hypothesis.

$P = P_1 \mid P_2$, we have that $\llparenthesis P_1 \mid P_2 \rrparenthesis = ((l)\mathtt{Par}\ \llparenthesis P_1 \rrparenthesis\ \llparenthesis P_2 \rrparenthesis\ l)$, and the thesis holds for the name l. For those in P_1 and P_2 by applying the inductive hypothesis we know that the thesis holds, as desired.

□

The next lemma is an invariant of our encoding. Essentially it states that each key l is freshly created and it is used only twice: by any evolution of the process itself and by the killer of the process that has generated it, for example the new key of a communication is only used by the continuation itself and by the `Mem` process.

Lemma A.9 *For any consistent configuration $\nu k.\, k : P$ and any $l \in \mathtt{n}(\llparenthesis k : P\rrparenthesis) \cap \mathcal{K}$, if $\llparenthesis k : P\rrparenthesis \Rightarrow R$, then one of the following sentences holds:*

1. *$R \equiv \mathbb{E}[\mathtt{addG}(\nu l.\, P_l \mid S)]$, with $S = (\mathtt{KillP}\ l\ l'\ h)$, $S = (\mathtt{KillP}\ l'\ l\ h)$, $S = (\mathtt{Mem}\, Y\ a\ \llparenthesis P\rrparenthesis\ l_1\ l\ l_2)$, $S = 0$ or S is obtained from the ones above via applications.*

2. *$R \equiv \mathbb{C}[\nu h.\, \mathbb{C}'[(l)P_l]h]$ where $\mathbb{C}'$ is a context as in Lemma A.8 or derived by one or more applications (closed under applications).*

3. *$R \equiv \mathbb{C}[\nu l.\, ((h)P_h)l]$*

4. *$R \equiv \mathbb{C}[\nu l.\, (((X\ c)c\langle\llparenthesis P\rrparenthesis\rangle)\ \llparenthesis Q\rrparenthesis\ c) \mid (c(Z)\triangleright Z\ l) \mid (\mathtt{Mem}\ (X\ c)c\langle\llparenthesis P\rrparenthesis\rangle)\ a\ \llparenthesis Q\rrparenthesis\ h\ l\ k)]$.*

Proof: By induction on the number of steps in $\llparenthesis k : P\rrparenthesis \Rightarrow S$. For the base case ($n = 0$) we have that $\llparenthesis \nu k.\, k : P\rrparenthesis = \nu k.\, \llparenthesis P\rrparenthesis k$. By Lemma A.8 we have that for any $l \in \mathtt{n}(\llparenthesis P\rrparenthesis) \cap \mathcal{K}$ we have $\llparenthesis P\rrparenthesis = \mathbb{C}'[(l)P_l]$, that is for any l we can write $\nu k.\, \llparenthesis P\rrparenthesis k \equiv \nu k.\, \mathbb{C}'[(l)P_l]k$, and condition (2) holds.

In the inductive case we have a case analysis according to which condition holds before the step. Let us consider the condition (1). We now proceed by case analysis on the structure of P_l. If $P_l = \llparenthesis P\rrparenthesis l$ we have to proceed by case analysis on the structure of P. If $P = 0$ then we have that the process $\mathbb{E}[\nu l.\, P_l \mid S] \rightharpoondown \mathbb{E}[\nu l.\, l\langle\mathtt{Nil}\rangle \mid \mathtt{Rew}\ l \mid S]$, as desired. If $P = a\langle Q\rangle$ then $\mathbb{E}[\nu l.\, \llparenthesis a\langle Q\rangle\rrparenthesis \mid S] \rightharpoondown \mathbb{E}[\nu l.\, \mathtt{Msg}\ a\ \llparenthesis Q\rrparenthesis\ l]$, as desired. The other cases are similar.

If $P_l = l\langle\llparenthesis P\rrparenthesis\rangle \mid \mathtt{Rew}\ l$, we have that $\mathbb{E}[\nu l.\, l\langle\llparenthesis P\rrparenthesis\rangle \mid \mathtt{Rew}\ l \mid S]$, can perform several reductions. If the context evolves by itself we can conclude. If $\mathtt{Rew}\ l$ is executed, then we are done. If the $\mathtt{Rew}\ l$ is already in its applied form then $\mathbb{E}[\nu l.\, l\langle\llparenthesis P\rrparenthesis\rangle \mid l(Z) \triangleright Z\ l \mid S] \hookrightarrow \mathbb{E}[\nu l.\, \llparenthesis P\rrparenthesis l \mid S]$, as desired. If the message on l is read by S there are two cases: either l contains the continuation of a trigger, or l contains a branch (left or right) of a `Par` process. In the first case we have that:

$$\begin{aligned}
&\mathbb{E}[\nu l.\, l\langle (\!|P|\!) \rangle \mid \texttt{Rew}\ l \mid (l(Z) \triangleright \texttt{Msg}\ a\ (\!|Q|\!)\ l_1 \mid \texttt{Trig}(X\ c)c\langle (\!|R|\!) \rangle\ a\ l_2) \mid S_1] \hookrightarrow \\
&\mathbb{E}[\nu l.\, \texttt{Rew}\ l \mid (\texttt{Msg}\ a\ (\!|Q|\!)\ l_1) \mid (\texttt{Trig}(X\ c)c\langle (\!|R|\!) \rangle\ a\ l_2)] \equiv \\
&\mathbb{E}'[\nu l.\, 0 \mid \texttt{Rew}\ l] = \mathbb{E}'[\nu l.\, \texttt{addG}(0)]
\end{aligned}$$

as desired. If it is a part of a parallel process, we have that in the context there is also a message on the channel h' such that:

$$\begin{aligned}
&\mathbb{E}[\nu l.\, l\langle (\!|P|\!) \rangle \mid \texttt{Rew}\ l] \equiv \\
&\mathbb{E}_1[\nu l.\, h\langle (\!|Q|\!) \rangle \mid l\langle (\!|P|\!) \rangle \mid (l(Z)|h(W) \triangleright k\langle (h)\texttt{Par}\ Z\ W\ h \rangle \mid \texttt{Rew}\ k)] \hookrightarrow \\
&\mathbb{E}_1[(\nu l.\, 0 \mid S_1) \mid k\langle (h)\texttt{Par}\ (\!|Q|\!)\ (\!|P|\!)\ h \rangle \mid \texttt{Rew}\ k] \equiv \mathbb{E}_1[\texttt{addG}(\nu l.\, 0 \mid S_1)]
\end{aligned}$$

as desired. If $P_l = \nu u.\, (\!|P|\!) l$, we have that $\mathbb{E}[\nu l.\, \texttt{addG}(\nu u.\, (\!|P|\!) l \mid S)] \equiv \mathbb{E}[\nu u.\, \nu l.\, \texttt{addG}((\!|P|\!) l \mid S)$, and the case is similar to $P_l = (\!|P|\!) l$.

If $P_l = \texttt{Msg}\ a\ (\!|Q|\!)\ l$ or $P_l = \texttt{Trig}\ Y\ a\ l$ we have that if the context evolves by itself the thesis banally follows. If the reduction is due to the P_l then it goes in its applied form that is still a P_l. Note that neither S nor the context can interact with P_l. We will consider the applied form of this case later on.

If $P_l = \texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ l$ we have that the context can evolve by itself and then condition (1) still holds, or the hole can execute by itself. If so, $\mathbb{E}[\nu l.\, (\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ l) \mid S] \hookrightarrow \mathbb{E}[\nu l, h, k.\, (\!|P|\!) h \mid (\!|Q|\!) k \mid (\texttt{KillP}\ h\ k\ l)]$, and we have that condition (1) is still satisfied by name l and by the new names h and k.

If $P_l = \texttt{KillP}\ h\ k\ l$ then we have that if the context execute by itself the condition (1) still holds. If P_l is not in its applied form, we have that after its application condition (1) still holds. The only way for the hole to interact with the context, is the presence of two messages on h and k and P_l is in its applied form. If so we have that:

$$\begin{aligned}
&\mathbb{E}[\nu l.\, h(W)|k(Z) \triangleright l\langle (h)\texttt{Par}\ W\ Z\ h \rangle] \equiv \\
&\mathbb{E}[\nu l.\, h(W)|k(Z) \triangleright l\langle (h)\texttt{Par}\ W\ Z\ h \rangle \mid h\langle (\!|P|\!) \rangle \mid \langle (\!|Q|\!) \rangle] \hookrightarrow \\
&\mathbb{E}[\nu l.\, l\langle (h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h \rangle \mid \texttt{Rew}\ l]
\end{aligned}$$

and condition (1) still holds.

Let us now consider its applied form. If $P_l = a\langle (\!|Q|\!), l \rangle \mid \texttt{KillM}\ a\ l$, we have that $\mathbb{E}[\nu l.\, P_l \mid S]$ can perform several reductions. If the context evolves by itself the thesis banally follows. If KillM is not in its applied form, then the thesis banally follows. If the KillM is in its applied form, then it can interact with the message. If so, we have that

$$\begin{aligned}&\mathbb{E}[\nu l.\, a\langle \llparenthesis Q \rrparenthesis, l\rangle \mid a(X,) \rhd l\langle (h)\mathtt{Msg}\; a\; \llparenthesis Q \rrparenthesis\; h\rangle \mid \mathtt{Rew}\; l \mid S] \hookrightarrow\\ &\mathbb{E}[\nu l.\, l\langle (h)\mathtt{Msg}\; a\; \llparenthesis Q \rrparenthesis\; h\rangle \mid (\mathtt{Rew}\; l) \mid S]\end{aligned}$$

and condition (1) still holds. If the message is read by the context, then the only cases are the presence of a `Trig` process or a `KillM`. In the first case we have that

$$\begin{aligned}&\mathbb{E}[\nu l.\, a\langle \llparenthesis Q \rrparenthesis, l\rangle \mid (\mathtt{KillM}\; a\; l)] \equiv\\ &\mathbb{E}_1[\nu l.\, a\langle \llparenthesis Q \rrparenthesis, l\rangle \mid \overline{\mathtt{t}} \mid (a(X,h)|\mathtt{t} \rhd \nu k, c.\, (((X\; c)c\langle \llparenthesis P \rrparenthesis\rangle)\; X\; c) \mid (c(Z) \rhd Z\; k) \mid\\ &(\mathtt{Mem}\; Y\; a\; X\quad h\; k\; l)) \mid (\mathtt{KillM}\; a\; l)] \hookrightarrow\\ &\mathbb{E}_1[\nu l, k, \mathtt{t}, c.\, (((X\; c)c\langle \llparenthesis P \rrparenthesis\rangle)\; \llparenthesis Q \rrparenthesis\; c) \mid c(Z) \rhd Z\; k \mid (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l\; k\; l_1) \mid S] \equiv\\ &\mathbb{E}_2[\nu k, l.\, (((X\; c)c\langle \llparenthesis Q \rrparenthesis\rangle)\; \llparenthesis P \rrparenthesis\; c) \mid c(Z) \rhd Z\; k \mid (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l\; k\; l_1) \mid S] \equiv\\ &\mathbb{E}_2[\nu l.\, (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l\; k\; l_1) \mid S]\end{aligned}$$

and the condition (1) still holds.

If $P_l = \nu\mathtt{t}.\,\overline{\mathtt{t}} \mid (\mathtt{t}|a(X,h) \rhd Q) \mid (\mathtt{KillT}\; Y\; \mathtt{t}\; l\; a)$, we have that $\mathbb{E}[\nu l.\, P_l \mid S]$, can perform several reductions. If the context evolves by itself then the thesis banally follows. If the `KillT` is applied then the thesis banally follows. If the `KillT` is in its applied form then we have that

$$\begin{aligned}&\mathbb{E}[\nu l.\, \nu\mathtt{t}.\,\overline{\mathtt{t}} \mid (\mathtt{t}|a(X,h) \rhd Q) \mid (\mathtt{t} \rhd l\langle (h)\mathtt{Trig}\; Y\; a\; h\rangle \mid \mathtt{Rew}\; l) \mid S] \hookrightarrow\\ &\mathbb{E}[\nu l.\, \nu\mathtt{t}.\, (\mathtt{t}|a(X,h) \rhd Q) \mid l\langle \llparenthesis P \rrparenthesis\rangle \mid \mathtt{Rew}\; l \mid S] \equiv\\ &\mathbb{E}[\nu l.\, \mathtt{addG}(\nu l.\, P_l \mid S)]\end{aligned}$$

as desired. If the process evolves via a communication $\rightarrow_f$ this implies that in the context there is a message on a of the form $a\langle \llparenthesis P \rrparenthesis, l\rangle$ (by Lemma A.7). Therefore,

$$\begin{aligned}&\mathbb{E}[\nu\mathtt{t}.\,\overline{\mathtt{t}} \mid (\mathtt{t}|a(X,h) \rhd Q) \mid \mathtt{KillT}\; Y\; \mathtt{t}\; l\; a] \equiv\\ &\mathbb{E}_1[\nu l, \mathtt{t}.\,\overline{\mathtt{t}} \mid a\langle \llparenthesis P \rrparenthesis, l_1\rangle \mid (\mathtt{t}|a(X,h) \rhd R) \mid \mathtt{KillT}\; Y\; \mathtt{t}\; l\; a] \rightarrow_f\\ &\mathbb{E}_1[\nu l, k, \mathtt{t}, c.\, (((X\; c)c\langle \llparenthesis Q \rrparenthesis\rangle)\; \llparenthesis P \rrparenthesis\; c) \mid c(Z) \rhd Z\; k \mid (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l_1\; k\; l) \mid S] \equiv\\ &\mathbb{E}_2[\nu k.\, (((X\; c)c\langle \llparenthesis Q \rrparenthesis\rangle)\; \llparenthesis P \rrparenthesis\; c) \mid c(Z) \rhd Z\; k \mid \nu l.\, (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l_1\; k\; l)] \equiv\\ &\mathbb{E}_3[\nu l.\, (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l_1\; k\; l) \mid S]\end{aligned}$$

and the condition (1) for name l, and condition (4) holds for the new name k.

If $P_l = \nu c.\, (Y\; \llparenthesis P \rrparenthesis\; c) \mid (c(Z) \rhd Z\; l) \mid (\mathtt{Mem}\; Y\; a\; \llparenthesis P \rrparenthesis\; l_1\; l\; l_2)]$, we have that

$\mathbb{E}[\nu l.\,(P_l \mid S)]$ can perform several reductions. If the Mem process is applied, we have that condition (1) is still satisfied. If the context evolves by itself, condition (1) still holds. If the reduction is the application of Y we have then

$$\begin{array}{l}\mathbb{E}[\nu l, c.\,(Y\ (\!|P|\!)\ c) \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ Y\ a\ (\!|P|\!)\ l_1\ l\ l_2)] \rightharpoondown \\ \mathbb{E}[\nu l, c.\,c\langle(\!|P_1|\!)\rangle \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ Y\ a\ (\!|P|\!)\ l_1\ l\ l_2)]\end{array}$$

and the condition (1) still holds. The symmetric case is equivalent.

If $P_l = \nu c.\,c\langle(\!|P|\!)\rangle \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ Y\ a\ (\!|P|\!)\ l_1\ l\ l_2)$, then if the Mem process is applied then condition (1) holds. Since c is a restricted channel, we have that the context cannot read from c. If the internal communication is performed (since c is a restricted) we have that

$$\begin{array}{l}\mathbb{E}[\nu l, c.\,c\langle(\!|P|\!)\rangle \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ Y\ a\ (\!|P|\!)\ l_1\ l\ l_2)] \hookrightarrow \\ \mathbb{E}[\nu l, c.\,(\!|P|\!) l \mid (\texttt{Mem}\ Y\ a\ (\!|P|\!)\ l_1\ l\ l_2)] \equiv \\ \mathbb{E}[\nu l.\,(P_l \mid S)]\end{array}$$

and condition (1) still holds.

If $P_l = \texttt{Mem}\ Y\ a\ (\!|P|\!)\ l\ k\ h$, we have that if the context evolves by itself or the application of the Mem is executed, then the thesis banally follows. If the Mem process is already in its applied form we have that it can interact just with the context via a message on k. Hence, we have

$$\begin{array}{l}\mathbb{E}[\nu l.\,k(Z) \triangleright (\texttt{Msg}\ a\ (\!|P|\!)\ l) \mid (\texttt{Trig}\ Y\ a\ h)] \hookrightarrow \\ \mathbb{E}[(\nu l.\,(\texttt{Msg}\ a\ (\!|P|\!)\ l)) \mid \texttt{Trig}\ Y\ a\ h] \equiv \\ \mathbb{E}_1[\nu l.\,(\texttt{Msg}\ a\ (\!|P|\!)\ l))]\end{array}$$

as desired. The other case of the memory $(\texttt{Mem}\ Y\ a\ (\!|P|\!)\ h\ k\ l)$ is equivalent.

For conditions (2), (3) and (4), we have that if the context evolves by itself they are still satisfied. If the context in condition (2) is an active one, we have to do a case analysis on the form of $\mathbb{C}'$. If the $\mathbb{C}' = \bullet$, then we have that $\mathbb{C}[\nu h.\,((l)P_l)h] \rightharpoondown \mathbb{C}[\nu h.\,P_h]$, the name l disappears and the lemma trivially holds. If $\mathbb{C}' = (l')\texttt{Msg}\ a\ \mathbb{C}''[\bullet]\ l'$ then we have that $\mathbb{C}[\nu h.\,((l')\texttt{Msg}\,a\,\mathbb{C}''[P_l]\ l')h] \rightharpoondown \mathbb{C}[\nu h.\,\texttt{Msg}\,a\,\mathbb{C}''[(l)P_l]\ h]$ and condition (2) still holds for name h. The other cases are similar for non applied context.

If condition (2) holds, and the context is an active one of the forms $\mathbb{C}[\nu h.\,\texttt{Msg}\ a\ \mathbb{C}'[(l)P_l]\ h]$, then we have that $\mathbb{C}[\nu h.\,\texttt{Msg}\ a\ \mathbb{C}'[(l)P_l]\ h] \rightharpoondown \mathbb{C}[\nu h.\,a\langle\mathbb{C}'[(l)P_l], h\rangle \mid \texttt{KillM}\,a\,h]$ and we note that condition (2) still holds for name l. The other cases are similar.

If condition (3) holds and the context is an active one, we have that

$\mathbb{C}[\nu l.\, ((h)P_h)l] \rightharpoondown \mathbb{C}[\nu l.\, P_l]$, and condition (1) holds for name l. If condition (4) holds, and the context is an execution one, we have that

$$\begin{aligned}
&\mathbb{C}[\nu l.\, (((X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ \llparenthesis Q \rrparenthesis\ c) \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ (X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ \llparenthesis Q \rrparenthesis\ h\ l\ k)] \rightharpoondown \\
&\mathbb{C}[\nu l.\, c\langle \llparenthesis P \rrparenthesis \rangle \{^{\llparenthesis Q \rrparenthesis}/_X\}] \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ (X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ \llparenthesis Q \rrparenthesis\ h\ l\ k)] = \\
&\mathbb{C}[\nu l.\, c\langle \llparenthesis P \rrparenthesis \{^{Q}/_X\} \rangle] \mid (c(Z) \triangleright Z\ l) \mid (\texttt{Mem}\ (X\ c)c\langle \llparenthesis P \rrparenthesis \rangle)\ a\ \llparenthesis Q \rrparenthesis\ h\ l\ k)]
\end{aligned}$$

By applying Lemma 5.8, and condition (1) holds for the name l. Naturally if the reduction involves the application of the `Mem` process then condition (1) for name l is still satisfied. □

The following Lemma is the equivalent of the Loop Lemma for the encoding. Essentially, it states that if there exist an execution from a translation of a consistent configuration M to a process P, then there exists a process and an execution such that from P we can reach Q and Q is *somehow* equivalent to the process P. Naturally we have to take into account all the garbage processes (killers and `Rew` processes) that the execution of $\llparenthesis M \rrparenthesis$ may generate.

Lemma 5.14. For any consistent configuration M, $\llparenthesis M \rrparenthesis \Rightarrow P$, if $P \hookrightarrow^* Q$ then exists Q' such that $Q \hookrightarrow^* Q'$ with $\texttt{nf}(P) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q'))$.

Proof: By induction on the number of steps of $P \hookrightarrow^* Q$. In the base case ($n = 0$) the thesis banally follows. We now consider one step, that is $P \hookrightarrow Q$. There are two cases to distinguish whether $\hookrightarrow$ is an application $\rightharpoondown$ or a non labelled communication $\rightarrow$. Let us consider the first case. We have that $P \rightharpoondown Q_1$ and $Q_1 \hookrightarrow^* Q$. By inductive hypothesis (on a shorter reduction) we have that there exists Q' such that $Q \hookrightarrow^* Q'$ and $\texttt{nf}(Q_1) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q'))$. By definition of normal form we have that $Q_1 \rightharpoondown^* \texttt{nf}(Q_1)$ and $P \rightharpoondown^* \texttt{nf}(P)$. Moreover, we have that $P \rightharpoondown Q_1$, and by using Lemma 5.12 we have that $P \rightharpoondown^* \texttt{nf}(Q_1)$, that is $\texttt{nf}(P) = \texttt{nf}(Q_1)$, and since $\texttt{nf}(Q_1) \equiv_{Ex} \texttt{nf}(\texttt{addG}(Q'))$ we can conclude.

If the step is a non labelled communication $\rightarrow$ we have three cases corresponding to three kinds of communications: the one involving a `Rew` process, the one involving a killer process and the one internal to the continuation of a trigger ($\rightarrow_c$). If the communication is via a `Rew` we have that $P \equiv \mathbb{E}[l\langle \llparenthesis P \rrparenthesis \rangle \mid l(Z) \triangleright Z\ l] \rightarrow \mathbb{E}[\llparenthesis P \rrparenthesis l]$ and by using Lemma 5.13 and Lemma A.14 we have that $\llparenthesis P \rrparenthesis l \hookrightarrow^* l\langle \llparenthesis P \rrparenthesis \rangle \mid \texttt{Rew}\ l \mid S$ with S a parallel composition of garbage processes. So, we have that $\mathbb{E}[\llparenthesis P \rrparenthesis l] \hookrightarrow^* \mathbb{E}[l\langle \llparenthesis P \rrparenthesis \rangle \mid \texttt{Rew}\ l \mid S] = Q$ with $\texttt{nf}(Q') \equiv \texttt{nf}(\texttt{addG}(P))$, as desired. If the communication is due to a kill process there are several cases to distinguish. If it is due to a `KillM`, then

$$\begin{aligned}
&\mathbb{E}[a\langle \llparenthesis P \rrparenthesis, l\rangle \mid (a(X, \backslash l) \triangleright l\langle (h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l]) \rightarrow \\
&\mathbb{E}[\langle (h)\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h)\rangle \mid \texttt{Rew}\ l] \rightharpoondown \\
&\mathbb{E}[l\langle (h)\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h)\rangle \mid l(Z) \triangleright Z\ l] \rightarrow \\
&\mathbb{E}[((h)\texttt{Msg}\ a\ \llparenthesis P \rrparenthesis\ h)l] \rightharpoondown^{*} \mathbb{E}[a\langle \llparenthesis P \rrparenthesis, l\rangle \mid (\texttt{KillM}\ a\ l)] = Q'
\end{aligned}$$

and we have that $\texttt{nf}(Q') = \texttt{nf}(P)$, as desired. If the communication is due to a `KillP` we have that

$$\begin{aligned}
&P \equiv \mathbb{E}[h\langle \llparenthesis P \rrparenthesis\rangle \mid l\langle \llparenthesis Q \rrparenthesis\rangle \mid (h(W)|l(Z) \triangleright k\langle (h)\texttt{Par}W\ Z\ h\rangle \mid \texttt{Rew}\ k)] \rightarrow \\
&\mathbb{E}[k\langle (h)\texttt{Par}\llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h\rangle \mid \texttt{Rew}\ k] \rightharpoondown \\
&\mathbb{E}[k\langle (h)\texttt{Par}\llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h\rangle \mid k(Z) \triangleright Z\ k] \rightarrow \\
&\mathbb{E}[((h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h)k] \rightharpoondown \mathbb{E}[\nu h, l.\, \llparenthesis P \rrparenthesis l \mid \llparenthesis Q \rrparenthesis h \mid (\texttt{KillP}\ l\ h\ k)]
\end{aligned}$$

and by using Lemma 5.13 and Lemma A.14 we have

$$\begin{aligned}
&\mathbb{E}[\nu h, l.\, \llparenthesis P \rrparenthesis l \mid \llparenthesis Q \rrparenthesis h \mid (\texttt{KillP}\ l\ h\ k)] \hookrightarrow^{*} \\
&\mathbb{E}[\nu h, l.\, l\langle \llparenthesis P \rrparenthesis\rangle \mid \texttt{Rew}\ l \mid S_1 \mid h\langle \llparenthesis Q \rrparenthesis\rangle \mid \texttt{Rew}\ h \mid S_2 \mid (\texttt{KillP}\ l\ h\ k)] = Q'
\end{aligned}$$

Now the only problem may be represented by the restriction on channels l and h, but by using Lemma A.9 we have that $P \equiv \mathbb{E}_1[\nu l, h.\, \texttt{addG}(h\langle \llparenthesis P \rrparenthesis\rangle \mid l\langle \llparenthesis Q \rrparenthesis\rangle \mid S)]$, and

$$\begin{aligned}
&\texttt{nf}(\mathbb{E}_1[\nu l, h.\, \texttt{addG}(h\langle \llparenthesis P \rrparenthesis\rangle \mid l\langle \llparenthesis Q \rrparenthesis\rangle \mid S \mid \texttt{Rew}\ h \mid \texttt{Rew}\ l)]) \equiv \\
&\texttt{nf}(\texttt{addG}(Q'))
\end{aligned}$$

as desired. If the communication is $\rightarrow_c$, we then have that

$$\begin{aligned}
&P \equiv \mathbb{E}[\nu c.\, c\langle \llparenthesis P \rrparenthesis\rangle] \mid c(Z) \triangleright Z\ k] \rightarrow_c \\
&\mathbb{E}[\nu c.\, \llparenthesis P \rrparenthesis k] \equiv_{Ex} \mathbb{E}[\nu c.\, c\langle \llparenthesis P \rrparenthesis\rangle] \mid c(Z) \triangleright Z\ k]
\end{aligned}$$

as desired. □

The following lemma is an invariant of our encoding. Essentially it states that the encoding never generates two messages on the same key channel, never generates two `KillP` processes waiting for the same rollbacks or never generates a `KillP` and a `Mem` waiting for the same rollback signal.

Lemma A.10 *For any $\rho\pi$ process R, if $\llparenthesis R \rrparenthesis l \Rightarrow P$ then the following conditions hold:*

1. $P \not\equiv \mathbb{C}[l_1\langle P\rangle \mid l_1\langle P_2\rangle]$, *with* $l_1 \in \mathcal{K}$.

2. $P \not\equiv \mathbb{C}[(\texttt{KillP}\ l_1\ l_2\ l) \mid (\texttt{KillP}\ l_3\ l_4\ l)]$, *with* $l, l_1, l_2, l_3, l_4 \in \mathcal{K}$ *and* $\{l_1, l_2\} \cap \{l_3, l_4\} = \emptyset$.

3. $P \not\equiv \mathbb{C}[(\texttt{KillP}\ l_1\ l_2\ l) \mid (\texttt{Mem}\ P\ a\ Q\ h\ l_3\ k)]$, *with* $l, l_1, l_2, l_3, h, k \in \mathcal{K}$ *and* $\{l_1, l_2\} \cap \{l_3\} = \emptyset$.

Proof: By simply inspecting the encoding of Figure 5.3.

For condition (1), let us note that messages on channels $l \in \mathcal{K}$ are generated by killer processes, by the continuation of a `Trig` processes or by `Nil` processes. We then proceed by structural induction of R. If the process R is a single process, that is a nil process, a message or a trigger, then there will be just a message on the channel l. If R is a restriction, then the key channel is passed to its sub term and no messages on l are generated by the restriction. If R is a parallel we can note that two new channels are created and given to the two sub processes and by inductive hypothesis the two sub processes will generate just one message each on them, while just a `KillP` is generated with the l channel. If the message is generate by a continuation of a `Trig` process we can see that in the continuation a new key channel is created and passed to the instantiated process, and by inductive hypothesis this process will generate just one message on that channel.

For condition (2), we can note that `KillP` is generated only by a `Par` process. We have that the process is instantiated with a fresh key, so there is no risk of producing another `KillP` reading on the same channel.

For condition (3) we can apply the same reasoning of conditions (1) and (2). □

Corollary 5.1. For any configuration M such that $\nu k.\, k : P \Rightarrow M$, if $\texttt{nf}(\llparenthesis M \rrparenthesis) \downarrow_a$ then $M \downarrow_a$.

Proof: We know that $\texttt{fn}(P) \cap \mathcal{K} = \emptyset$, and that also $\texttt{fn}(M) \cap \mathcal{K} = \emptyset$. Now by using Lemma A.6 we have that $\texttt{fn}(\texttt{nf}(\llparenthesis M \rrparenthesis)) \cap \mathcal{K} = \emptyset$. And now, we can conclude by using Lemma 5.17. □

Next lemma states that if a process, derived by the encoding, contains a message on a key channel l, then the process contains also the `Rew` l process.

Lemma A.11 (`Rew` Invariant) *If* $\llparenthesis \nu k.\, k : P \rrparenthesis \Rightarrow P'$ *with* $P' \equiv \mathbb{C}[l\langle R\rangle]$ *then* $P' \equiv \mathbb{C}'[l\langle R\rangle \mid S]$ *with* $S = \texttt{Rew}\ l$ *or* $S = l(Z) \triangleright Z\ l$ *, where* $\mathbb{C}, \mathbb{C}'$ *are n-ary contexts.*

Proof: By induction on the number of steps in $\Rightarrow$. The base case is when $\llparenthesis \nu k.\, k : P \rrparenthesis = P'$, and by definition we have that $P' = \nu k.\, \llparenthesis P \rrparenthesis k$, and we proceed by structural induction on P.

$P = 0$ **:** we have that $\llparenthesis P \rrparenthesis = (l)(\langle \mathtt{Nil} \rangle \mid \mathtt{Rew}\, l)$, as desired.

$P = a\langle Q \rangle$ **:** we have that $\llparenthesis a\langle Q \rangle \rrparenthesis = (l)(\mathtt{Msg}\ a\ \llparenthesis Q \rrparenthesis\ l) = (l)((a\ X\ l)a\langle X, l\rangle \mid ((a,l)a(X, \backslash l) \triangleright l\langle R \rangle \mid \mathtt{Rew}\ l)\ a\ l)\ a\ \llparenthesis Q \rrparenthesis\ l)$, and we can conclude by inductive hypothesis on Q, as desired.

$P = a(X) \triangleright Q$ **:** be $Y = (X\ c)c\langle Q \rangle$, we have that $\llparenthesis a(X) \triangleright Q \rrparenthesis = (l)(\mathtt{Trig}\, Y\ a\ l)$ $= (l)((Y\ a\ l)\nu \mathtt{t}.\, \overline{\mathtt{t}} \mid a(X, \backslash h)|\mathtt{t} \triangleright R_1 \mid \mathtt{KillT}\ Y\ \mathtt{t}\ l\ a)\ Y\ a\ l) =$ $(l)((Y\ a\ l)\nu \mathtt{t}.\, \overline{\mathtt{t}} \mid a(X, \backslash h)|\mathtt{t} \triangleright R_1 \mid ((Y\ t\ \ l\ a)t \triangleright l\langle R \rangle \mid \mathtt{Rew}\, l)\ Y\ \mathtt{t}\ l\ a)\ Y\ a\ l)$, as desired.

$P = \nu a.\, Q$ **:** we have that $\llparenthesis \nu a.\, Q \rrparenthesis = (l)\nu a.\, \llparenthesis Q \rrparenthesis$, and by inductive hypothesis we have that $\llparenthesis Q \rrparenthesis \equiv \mathbb{C}[l\langle R \rangle \mid \mathtt{Rew}\, l]$ and so $\llparenthesis P \rrparenthesis \equiv (l)\mathbb{C}'[l\langle R \rangle \mid \mathtt{Rew}\, l]$, and we can conclude by inductive hypothesis on Q, as desired.

$P = Q_1 \mid Q_2$ **:** we have that $\llparenthesis P \rrparenthesis = (l)(\mathtt{Par} \llparenthesis Q_1 \rrparenthesis\ \llparenthesis Q_2 \rrparenthesis\ l)$ and by using inductive hypothesis we have that $\llparenthesis Q_i \rrparenthesis = \mathbb{C}_i[l_i\langle R_i \rangle \mid \mathtt{Rew}\ l_i]$, and so $\llparenthesis P \rrparenthesis \equiv (l)(\mathtt{Par}\ \mathbb{C}_1[l_1\langle R_1 \rangle \mid \mathtt{Rew}\, l_1]\ \ \mathbb{C}_2[l_2\langle R_2 \rangle \mid \mathtt{Rew}\, l_2]\ l)$, as desired.

In the inductive case we have that $P \Rightarrow P'' \rightarrow P'$ with $P' \equiv \mathbb{C}[l\langle R \rangle]$. We proceed by case analysis on $P'' \rightarrow P'$. If $\rightarrow = \rightharpoondown$ then we have to distinguish whether the abstraction is on a process or on a channel.

If the abstraction is on a process this implies that $P' \equiv \mathbb{E}[Q\{{}^{V}/_{X}\}]$ and then $P'' \equiv \mathbb{E}[(X)Q\ V]$. Now we have to consider different cases on the position of $l\langle R \rangle$ in P', and for simplicity we will consider just one instance of $l\langle R \rangle$ at time. We have three cases: $l\langle R \rangle$ is inside the context $\mathbb{E}$, it is inside in the process Q or inside the value V. In the first case we can write $P' \equiv \mathbb{E}[\mathbb{C}[l\langle R \rangle] \mid Q\{{}^{V}/_{X}\}]$, but this implies that $P'' \equiv \mathbb{E}[\mathbb{C}[l\langle R \rangle] \mid (X)Q\ V]$ and by inductive hypothesis $P'' \equiv \mathbb{E}[\mathbb{C}[l\langle R \rangle \mid S] \mid (X)Q\ V] \rightharpoondown \mathbb{E}[\mathbb{C}[l\langle R \rangle \mid S] \mid Q\{{}^{V}/_{X}\}] \equiv P'$ as desired. In the second case we have $P'' \equiv \mathbb{E}[C[l\langle R \rangle]\{{}^{V}/_{X}\}] = \mathbb{E}[\mathbb{C}'[l\langle R\{{}^{V}/_{X}\}\rangle]]$, but this implies that $P'' \equiv \mathbb{E}[(X)\mathbb{C}[l\langle R \rangle]\ V] \equiv \mathbb{E}[(X)\mathbb{C}[l\langle R \rangle \mid S]\ V] \rightharpoondown \mathbb{E}[\mathbb{C}[l\langle R \rangle \mid S]\{{}^{V}/_{X}\}] = \mathbb{E}[\mathbb{C}[l\langle R\{{}^{V}/_{X}\}\rangle \mid S\{{}^{V}/_{X}\}]]$. We have now to show the effect of the substitution on S, if $S = \mathtt{Rew}\, l$ then $S\{{}^{V}/_{X}\} = S$ else if $S = l(Z) \triangleright Z\ l$ then again $S\{{}^{V}/_{X}\} = S$. So, $P'' \rightharpoondown \mathbb{E}[\mathbb{C}[l\langle R\{{}^{V}/_{X}\}\rangle \mid S]] \equiv P'$ as desired. Note that in this case there is no way that the application we are considering is of the form $\mathtt{Rew}\, l$. If the considered message appears in the value V then we have that $P'' \equiv \mathbb{E}[Q\{{}^{\mathbb{C}[l\langle R \rangle]}/_{X}\}]$, and this implies that $P'' \equiv \mathbb{E}[(X)Q\ \mathbb{C}[l\langle R \rangle]]$ and by inductive hypothesis $P'' \equiv \mathbb{E}[(X)Q\ \mathbb{C}'[l\langle R \rangle \mid S]] \rightharpoondown \mathbb{E}[Q\{{}^{\mathbb{C}'[l\langle R \rangle|S]}/_{X}\}] \equiv P'$, as desired.

If the abstraction is on a channel this implies that $P' \equiv \mathbb{E}[Q\{{}^{l'}/_{h}\}]$ and then $P'' \equiv \mathbb{E}[(h)Q\ l']$. As in the above case, there are three positions in which $l\langle R\rangle$ may appear in P'. If we have that $P' \equiv \mathbb{E}[\mathbb{C}[l\langle R\rangle] \mid Q\{{}^{l'}/_{h}\}]$ this implies that $P'' \equiv \mathbb{E}[\mathbb{C}[l\langle R\rangle] \mid (h)Q\ l']$. Now if $((h)Q\ l') \neq (\mathtt{Rew}\ l)$ we have that by inductive hypothesis

$$P'' \equiv \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid S] \mid (h)Q\ l'] \rightharpoondown \\ \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid S] \mid Q\{{}^{l'}/_{h}\}] \equiv P'$$

as desired. If the process Q is the $\mathtt{Rew}$ of the message $l\langle R\rangle$ we are considering, then $P'' \equiv \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid \mathtt{Rew}\ l] \rightharpoondown \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid l(Z) \triangleright Z\ l] \equiv P'$, as desired. If the $\mathtt{Rew}\ l$ is a different one then by applying the inductive hypothesis

$$P'' \equiv \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid S] \mid (\mathtt{Rew}\ l)] \rightharpoondown \\ \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid S] \mid l(Z) \triangleright Z\ l] \equiv P'$$

as desired. If the message we are considering is present in the abstraction then we have that $P' \equiv \mathbb{E}[\mathbb{C}[l\langle R\rangle]\{{}^{l'}/_{h}\}] = \mathbb{E}[\mathbb{C}''[l\langle R\{{}^{l'}/_{h}\}\rangle]]$ where $\mathbb{C}''$ is the resulting context after the substitution. Hence $P'' \equiv \mathbb{E}[(h)\mathbb{C}[l\langle R\rangle]\ l']$, but by inductive hypothesis we have that

$$P'' \equiv \mathbb{E}[(h)\mathbb{C}'[l\langle R\rangle \mid S]\ l'] \rightharpoondown \mathbb{E}[\mathbb{C}'[l\langle R\rangle \mid S]]\{{}^{l'}/_{h}\} = \\ \mathbb{E}[\mathbb{C}'''[l\langle R\{{}^{l'}/_{h}\}\rangle \mid S\{{}^{l'}/_{h}\}]] \equiv P'$$

as desired. Note that the substitution on S has effect only if the bound variable of the abstraction is l, but this will implies that in the whole process $\mathbb{C}[l\langle R\rangle]$ all the free occurrences of l will be substituted by l', against the hypothesis that $l\langle R\rangle$ occurs in the resulting process.

If the reduction is a communication, as in the case of abstraction, we have to consider all the different places in which the considered message can appear. We have to distinguish two cases: whether the subject of the communication is the channel l or not. If not, as in the case of the abstraction, we have to consider three cases depending on the position of the considered $l\langle R\rangle$. If it is in the context then we have that $P' \equiv \mathbb{E}[Q\{{}^{P}/_{X}\}]$ and then $P'' \equiv \mathbb{E}[a\langle P\rangle \mid a(X) \triangleright Q \mid \mathbb{C}[l\langle R\rangle]]$ and by inductive hypothesis we have $P'' \equiv \mathbb{E}[a\langle P\rangle \mid a(X) \triangleright Q \mid \mathbb{C}'[l\langle R\rangle \mid S]] \rightarrow \mathbb{E}[Q\{{}^{P}/_{X}\} \mid \mathbb{C}'[l\langle R\rangle \mid S]] \equiv P'$, as desired. If it is present in the content of the message then we have $P' \equiv \mathbb{E}[Q\{{}^{\mathbb{C}[l\langle R\rangle]}/_{X}\}]$ and then $P'' \equiv \mathbb{E}[a\langle \mathbb{C}[l\langle R\rangle]\rangle \mid a(X) \triangleright Q]$ and by inductive hypothesis $P'' \equiv \mathbb{E}[a\langle \mathbb{C}'[l\langle R\rangle \mid S]\rangle \mid a(X) \triangleright Q] \rightarrow \mathbb{E}[Q\{{}^{\mathbb{C}'[l\langle R\rangle \mid S]}/_{X}\}]$, as desired. The other case is similar. □

Next lemma states that if a process, derived by the encoding, contains a message on a channel $a \in \mathcal{N}$, then the process contains also its own `KillM` process. That is, messages on normal channel are always generated along with their own killers.

Lemma A.12 (`KillM` Invariant) *If* $(\!|\nu k.\, k : P|\!) \Rightarrow P'$ *with* $P' \equiv \mathbb{C}[a\langle(\!|P|\!), l\rangle]$ *with* $a \in \mathcal{N}$ *then* $P' \equiv \mathbb{C}'[a\langle(\!|P|\!), l\rangle \mid S]$ *with* $S = (\mathtt{KillM}\ a\ l)$ *or* $S = (a(X, \backslash l) \triangleright l\langle(h)\mathtt{Msg}\ a\ X\ h\rangle \mid \mathtt{Rew}\ l)$ *, where* $\mathbb{C}, \mathbb{C}'$ *are n-ary contexts.*

Proof: By simply inspecting the encoding of Figure 5.3 we note that the sole place where a message of the form $a\langle(\!|P|\!), l\rangle$ is generated is in the `Msg` process that generates also its corresponding `KillM`. □

Lemma A.13 (Input key invariant) *For each ρπ process R we have:*

1. $(\!|R|\!)l \not\Rightarrow \mathbb{E}[l(X) \triangleright P]$*, unless the trigger has been generated by an application of* `Rew` l*.*
2. $(\!|R|\!)l \not\Rightarrow \mathbb{E}[l(X)|l'(W) \triangleright P]$*.*
3. $(\!|R|\!)l \not\Rightarrow \mathbb{E}[l'(X) \triangleright P \mid l'(X) \triangleright Q]$ *for each* $l' \in \mathcal{K}$ *and* $l' \neq l$*.*

Proof: (1) We now show that $(\!|R|\!)l \not\Rightarrow \mathbb{E}[l(X) \triangleright P]$ in a number of steps that is less or equal than n, so we proceed by induction on n. The base case $(n = 0)$ is banally verified since $(\!|R|\!)l$ it is an application and not a trigger on l. Now we reason by contradiction. Let us suppose that $(\!|R|\!)l \Rightarrow \mathbb{E}[l(X) \triangleright P]$ in n steps. By looking at the encoding in Figure 5.3 we note that there is just one place in which such a trigger can be generated: `Mem`. If the trigger is generated by a `Mem` then it has the form of $k(Z) \triangleright P$. Since a memory process can be generated just by a `Trig` process, we can note that the `Mem` process is invoked with a channel k that is fresh (generated in the continuation of a trigger), hence we have that $k \neq l$, as desired.

(2) The base case is banally verified. As in (1) we reason by contradiction. Let us suppose that $(\!|R|\!)l \Rightarrow \mathbb{E}[l(X)|l'(Z) \triangleright P]$ in n steps. By looking at the encoding there is just one place where such trigger can be generated: `KillP`. Since this killer is generated by a `Par` process we can note that the `KillP` process is invoked with two new fresh channels, and hence they are different from l.

(3) We use the same proof strategy as (1) and (2), by noting that such a triggers on channels $l' \in \mathcal{K}$ with $l \neq l$ are generated by `KillP` and `Mem` process, but again these processes are invoked respectively in a `Par` process or a `Trig` process with fresh new channels.

□

The following lemma states that messages on channels $l \in \mathcal{K}$ carry translation of processes with no restrictions.

Lemma A.14 *For each $\rho\pi$ process R, $\llparenthesis R \rrparenthesis l \Rightarrow \mathbb{C}[l\langle Q\rangle]$, then $Q = \llparenthesis Q' \rrparenthesis$ with Q' not containing restrictions.*

Proof: For the base case we proceed by structural induction on R.

$R = 0$ **:** by definition $\llparenthesis 0 \rrparenthesis h = \mathtt{Nil} = ((l)(l\langle \mathtt{Nil}\rangle \mid (\mathtt{Rew}\ l)))h$, and the property is banally verified.

$R = a\langle P\rangle$ **:** by definition

$$\begin{aligned}
&\llparenthesis a\langle P\rangle \rrparenthesis h = (l)((a\ X\ l)\mathtt{Msg}\ a\ X\ l)h = \\
&(l)((a\ X\ l)(a\langle X, \backslash l\rangle \mid (\mathtt{KillM}\ a\ l))\ a\ \llparenthesis P \rrparenthesis\ l)h = \\
&(l)(((a\ X\ l)(a\langle X, \backslash l\rangle \mid (((a\ l)a(X, \backslash l) \triangleright l\langle (h)\mathtt{Msg}\ a\ X\ h\rangle \mid \\
&(\mathtt{Rew}\ l))\ a\ l))a\ \llparenthesis P \rrparenthesis\ l)h
\end{aligned}$$

as desired.

$R = a(X) \triangleright Q$ **:** by definition

$$\begin{aligned}
&\llparenthesis a(X) \triangleright Q \rrparenthesis h = (l)(\mathtt{Trig}\ (X\ c)c\langle \llparenthesis P \rrparenthesis\rangle\ a\ l)h = \\
&(l)((Y\ a\ l)(\nu \mathtt{t}.\, \overline{\mathtt{t}} \mid \mathtt{t}|a(X, h)\triangleright \mid \nu k, c.\, (Y\ X\ c) \mid c(Z) \triangleright Z\ k \mid \\
&\quad (\mathtt{Mem}\ Y\ a\ X\ h\ k\ l)) \mid \mathtt{KillT}\ Y\ \mathtt{t}\ a\ c)\ (X\ c)c\langle \llparenthesis P \rrparenthesis\rangle\ a\ l)h = \\
&(l)((Y\ a\ l)(\nu \mathtt{t}.\, \overline{\mathtt{t}} \mid \mathtt{t}|a(X, h)\triangleright \mid \nu k, c.\, (Y\ X\ c) \mid c(Z) \triangleright Z\ k \mid \\
&\quad (((Y\ a\ X\ h\ k\ l)k(Z) \triangleright (\mathtt{Msg} a\ X\ h) \mid (\mathtt{Trig}\ Y\ a\ l))\ Y\ a\ X\ h\ k\ l)) \mid \\
&\quad (((Y\ t\ l\ a)\mathtt{t} \triangleright l\langle (h)\mathtt{Trig}\ Y\ l\ a\rangle)\ Y\ \mathtt{t}\ a\ c)\ (X\ c)c\langle \llparenthesis P \rrparenthesis\rangle\ a\ l)h
\end{aligned}$$

and by applying the inductive hypothesis on $Z\ k$ we can conclude. Let us note that the input process generated by the `Mem` process reads on a fresh new channel k.

$R = \nu a.\, P$ **:** by definition $\llparenthesis \nu a.\, P \rrparenthesis h = (l)(\nu a.\, \llparenthesis P \rrparenthesis)h$ and we can conclude by inductive hypothesis on P.

$R = P_1 \mid P_2$ **:** by definition

$$\begin{array}{l}(\!|P_1 \mid P_2|\!)h = (l)(\texttt{Par}\ (\!|P_1|\!)\ (\!|P_2|\!)\ l)h = \\ (l)(((X\ Y\ l)\nu h,k.\, X\ h \mid Y\ k \mid (\texttt{KillP}\ h\ k\ l))\ (\!|P_1|\!)\ (\!|P_2|\!)\ l)h = \\ (l)(((X\ Y\ l)\nu h,k.\, X\ h \mid Y\ k \mid \\ \quad (((h\ k\ l)h(W)|k(Z) \triangleright l\langle(l)(\texttt{Par}\ W\ Z\ l)\rangle)\ h\ k\ l))\ (\!|P_1|\!)\ (\!|P_2|\!)\ l)h\end{array}$$

Now by using inductive hypothesis on $X\ h$ and $Y\ k$ and since k and h and since h and k are new channels we can conclude.

For the inductive case, we have that either the message we are considering disappear, and then the thesis banally holds, or the message remains in the context and we can conclude directly by applying the inductive hypothesis.

□

A.3 Congruence

In this section we will prove that both addG and $\equiv_{Ex}$ are weak bf barbed bisimulation. The proof strategy is similar, we will start proving one by one that single applications of addG or $\equiv_{Ex}$ are bisimulation. Then we will compose the single results by transitivity.

The following Lemmas show how the processes added by addG function does not add any unexpected behavior, that is they can be considered as the zero process.

Lemma A.15 *Let T_l be a process of the form $T_l = \texttt{Rew}\ l$ or $T_l = (l(Z) \triangleright Z\ l)$. Let $\mathcal{R} = \{(\mathbb{C}[T_l], \mathbb{C}[0])\}$. The the relation $\mathcal{R}$ is a back and forth barbed bisimulation.*

Proof: Let us start with barbs. Since $\texttt{Rew}\ l$ does show any barbs, we have that the only barbs shown by $\mathbb{C}[\texttt{Rew}\ l]$ are those of the context. So barbs are banally matched by the process $\mathbb{C}[0]$, and vice versa.

Let us consider the reductions. We have that if $\mathbb{C}[T_l]$ reduces then it is because the context reduces by itself, because the hole process reduces by itself or because of an interaction between the context and the hole process. In the first case the reduction is banally matched by the process $\mathbb{C}[0]$. The second case implies that $T_l = \texttt{Rew}\ l$ and so $\mathbb{C}[\texttt{Rew}\ l] \rightharpoondown \mathbb{C}[l(Z) \triangleright Z\ l]$, and this step is matched by a null execution of $\mathbb{C}[0]$, and we remain in the same relation. The third case implies that $T_l = (l(Z) \triangleright Z\ l)$ and the presence of a message in the context of the form $l\langle(\!|P|\!)\rangle$. Hence, we have that $\mathbb{C}[(l(Z) \triangleright Z\ l)] \equiv \mathbb{C}'[l\langle(\!|P|\!)\rangle \mid (l(Z) \triangleright Z\ l)]$ and $\mathbb{C}[0] \equiv \mathbb{C}'[l\langle(\!|P|\!)\rangle]$. By Lemma A.11 we know $\mathbb{C}'[l\langle(\!|P|\!)\rangle] \equiv \mathbb{C}''[l\langle(\!|P|\!)\rangle \mid T_l]$ but also on the other side we have that $\mathbb{C}'[l\langle(\!|P|\!)\rangle \mid (l(Z) \triangleright Z\ l)] \equiv \mathbb{C}''[l\langle(\!|P|\!)\rangle \mid (l(Z) \triangleright Z\ l) \mid T_l]$. So, $\mathbb{C}''[l\langle(\!|P|\!)\rangle \mid (l(Z) \triangleright Z\ l) \mid T_l] \to \mathbb{C}''[(\!|P|\!)l \mid T_l]$ on the other side we have that this step can be matched by $\mathbb{C}''[l\langle(\!|P|\!)\rangle \mid T_l] \hookrightarrow^* \mathbb{C}''[(\!|P|\!)l]$, and we are still in the same relation since $\mathbb{C}''[(\!|P|\!)l \mid T_l] \equiv \mathbb{C}'''[T_l]$ and $\mathbb{C}''[(\!|P|\!)l] \equiv \mathbb{C}'''[0]$, with $(\mathbb{C}'''[T_l], \mathbb{C}'''[0]) \in \mathcal{R}$, as desired. □

Lemma A.16 *Let $T(l,a)$ be a process of the form $T(l,a) = (\texttt{KillM}\ a\ l)$ or $T(l,a) = (a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)$. Let $\mathcal{R} = \{(\mathbb{C}[T(l,a)], \mathbb{C}[0])\}$. The relation $\mathcal{R}$ is a weak bf barbed bisimulation.*

Proof: Let us consider the barbs. Since a process of the form $(\texttt{KillM}\ a\ l)$ or $(a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)$ does not show any barbs, then the only barbs shown by $\mathbb{C}[T(l,a)]$ are those of the context $\mathbb{C}$. Hence, the barbs are banally matched by the process $\mathbb{C}[0]$ (and vice versa).

Let us consider the reductions. If the context evolves by itself, this reduction is banally matched. If $T(l,a) = (\texttt{KillM}\ a\ l)$ then the only possible reduction is the application, so

$$\begin{aligned}&\mathbb{C}[(\texttt{KillM}\ a\ l)] \rightharpoondown \\ &\mathbb{C}[(a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)]\end{aligned}$$

and this reduction is matched by a zero reduction by the process $\mathbb{C}[0]$. If $\mathbb{C}[(a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)]$ evolves by the interaction of the hole process and the context, it is because of the presence of a message of the form $a\langle(\!|P|\!), \backslash l\rangle$ in the context. Hence

$$\begin{aligned}&\mathbb{C}[(a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)] \equiv \\ &\mathbb{C}'[a\langle(\!|P|\!), \backslash l\rangle \mid (a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)]\end{aligned}$$

This implies that also $\mathbb{C}[0] \equiv \mathbb{C}'[a\langle(\!|P|\!), \backslash l\rangle]$ and by Lemma A.12 we know that $\mathbb{C}'[a\langle(\!|P|\!), \backslash l\rangle] \equiv \mathbb{C}''[a\langle(\!|P|\!), \backslash l\rangle \mid S]$ with $S = (\texttt{KillM}\ a\ l)$ or $S = (a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)$. Then

$$\begin{aligned}&\mathbb{C}'[a\langle(\!|P|\!), \backslash l\rangle \mid (a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l)] \equiv \\ &\mathbb{C}''[a\langle(\!|P|\!), \backslash l\rangle \mid (a(X, \backslash l) \triangleright l\langle(h)\texttt{Msg}\ a\ X\ h\rangle \mid \texttt{Rew}\ l) \mid S] \to \\ &\mathbb{C}''[l\langle(h)\texttt{Msg}\ a\ (\!|P|\!)\ h\rangle \mid \texttt{Rew}\ l) \mid S] \equiv \mathbb{C}'''[S]\end{aligned}$$

and on the other side we have that

$$\mathbb{C}''[a\langle (\!|P|\!), \backslash l\rangle \mid S] \hookrightarrow^* \mathbb{C}''[l\langle (h)\mathtt{Msg}\ a\ (\!|P|\!)\ h\rangle \mid \mathtt{Rew}\ l] \equiv \mathbb{C}'''[0]$$

and we remain in the same relation, as desired. □

Lemma A.17 *Let $\mathcal{R} = \{(\mathbb{C}[\nu\mathtt{t}.\,(a(X,k)|\mathtt{t} \triangleright Q)], \mathbb{C}[0])\}$. The relation $\mathcal{R}$ is a weak bf barbed bisimulation.*

Proof: Let us consider the barbs. Since the process $\nu\mathtt{t}.\,(a(X,k)|\mathtt{t} \triangleright Q)$, the only barbs shown by $\mathbb{C}[\nu\mathtt{t}.\,(a(X,k)|\mathtt{t} \triangleright Q)]$ are those of $\mathbb{C}[\_]$. Hence, the barbs are banally matched by the process $\mathbb{C}[0]$ and vice-versa.

Let us consider reductions. Since the name $\mathtt{t}$ is restricted, this implies that the context cannot interact with the process $(a(X,k)|\mathtt{t} \triangleright Q)$ even if there is a message on a in the context. Hence, if $\mathbb{C}[\nu\mathtt{t}.\,(a(X,k)|\mathtt{t}{\triangleright}Q)] \to$ it is because the context evolved by itself, that is $\mathbb{C}[\nu\mathtt{t}.\,(a(X,k)|\mathtt{t}{\triangleright}Q)] \to \mathbb{C}'[\nu\mathtt{t}.\,(a(X,k)|\mathtt{t}{\triangleright}Q)]$ and this reduction banally is matched by $\mathbb{C}[0] \to \mathbb{C}'[0]$ (and vice-versa), and we are still in the same relation. □

Lemma A.18 *Let $T((\!|P|\!), a, l, c)$ be a process of one of the following form:*

$$T((\!|P|\!), a, l, c) = \nu c, \mathtt{t}.\,(\mathtt{KillT}\ ((X)c\langle(\!|P|\!)\rangle)\ \mathtt{t}\ l\ a)$$
$$T((\!|P|\!), a, l, c) = \nu c, \mathtt{t}.\,(\mathtt{t} \triangleright l\langle (h)\mathtt{Trig}\ ((X)c\langle(\!|P|\!)\rangle)\ a\ l\rangle \mid \mathtt{Rew}\ l)$$

Let $\mathcal{R} = \{(\mathbb{C}[T((\!|P|\!), a, l, c)], \mathbb{C}[0])\}$. The relation $\mathcal{R}$ is a weak barbed bf bisimulation.

Proof: By just noting that the two forms of $T((\!|P|\!), a, l, c)$ do not add any barb. For the reductions, let us note that if $\mathbb{C}[T((\!|P|\!), a, l, c)]$ reduces it is because the hole reduced by itself (and this step can be exactly mimicked) or because the hole process reduced by itself. Since the name $\mathtt{t}$ is restricted, then hole and context cannot interact each other. If the hole process executes by its own, then the sole reduction can be $\mathbb{C}[\nu c, \mathtt{t}.\,(\mathtt{KillT}\ ((X)c\langle(\!|P|\!)\rangle)\ \mathtt{t}\ l\ a)]$ $\rightharpoondown \mathbb{C}[\nu c, \mathtt{t}.\,(\mathtt{t} \triangleright l\langle (h)\mathtt{Trig}\ ((X)c\langle(\!|P|\!)\rangle)\ a\ l\rangle \mid \mathtt{Rew}\ l)]$, and this reduction can be matched by a zero step execution of the process $\mathbb{C}[0]$, and we remain in the same relation. If the reductions are due to the context then the same reduction can be mimicked (and vice-versa), and we are in the same relation. □

Now we are able to compose all the previous Lemmas on the various production of `addG` to show that `addG` is a weak backward and forward barbed bisimulation.

Lemma 5.23. For any HOπ^+ processes P, Q, the relation $\mathcal{R} = \{(P, Q) \mid P \equiv Q\}$ is a weak bf barbed bisimulation.

Proposition 5.1. For any HOπ^+ process $P \equiv \nu\tilde{a}.\, P'$, the relation $\mathcal{R} = \{(\nu\tilde{a}.\, P',\ \nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, Q)\}$ with Q a parallel composition of processes as in Definition 5.4 is a weak bf barbed bisimulation.

Proof: By induction on the number of processes contained in the parallel composition Q. The base case, with $Q = 0$ banally reduces to the identity since we can can garbage collect via structural equivalence names contained in $\tilde{b}$ (and structural equivalence is a weak bf barbed bisimulation by Lemma 5.23). In the inductive case, we do a case analysis on the last process Q_n of the parallel composition.

$Q_n = \texttt{Rew}\ l$**:** by Lemma A.15 we know that $\mathbb{C}[Q_n] \overset{\circ}{\approx}_c \mathbb{C}[0]$ and so we have that $\nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, \prod_{i=1..n} Q_i) \overset{\circ}{\approx}_c \nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, \prod_{i=1..n-1} Q_i)$. By inductive hypothesis we have that $\nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, \prod_{i=1..n-1} Q_i) \overset{\circ}{\approx}_c \nu\tilde{a}.\, P'$ and by transitivity we have $\nu\tilde{a}.\, (P' \mid \nu\tilde{b}.\, \prod_{i=1..n} Q_i) \overset{\circ}{\approx}_c \nu\tilde{a}.\, P'$.

$Q_n = \texttt{KillM}\ a\ l$**:** by using Lemma A.16 and by using the same reasoning of the first case.

$Q_n = \nu\texttt{t}.\, (a(X,k)|\texttt{t} \triangleright Q)$**:** by using Lemma A.17 and by using the same reasoning of the first case.

$Q_n = \nu c, \texttt{t}.\, (\texttt{KillT}\ ((X)c(\!|P|\!))\ t\ l\ a)$**:** by using Lemma A.18 and by using the same reasoning of the first case.

□

The following Lemmas are meant to prove that the relation $\equiv_{Ex}$ is a weak bf barbed bisimulation. We will refer to $\mathbb{C}$ as a multi holes context.

Lemma A.19 *Axiom Ex.C and its instantiation are correct with respect to weak bf barbed bisimulation.*

Proof: Let $\mathcal{R}' = \{((\texttt{KillP}\ l\ h\ k), (\texttt{KillP}\ h\ l\ k)) \mid h, l, k \in \mathcal{K}\} \cup \{((a(X)|b(Y) \triangleright R), (b(Y)|a(X) \triangleright R)) \mid a, b \in \mathcal{N}\}$. Let $\mathcal{R} = \{(\mathbb{C}[P_1, .., P_n], \mathbb{C}[Q_1, .., Q_n]) \mid (P_i, Q_i) \in \mathcal{R}' \wedge i = 1..n \wedge n \in \mathbb{N} \wedge X, Y \in \mathcal{V} \wedge R \in \mathcal{P}\}$. We now show that the relation $\mathcal{R}$ is a weak bf barbed bisimulation.

Let us consider the barbs. We can directly note that neither $\texttt{KillP}\ l\ h\ k$ nor $\texttt{KillP}\ h\ l\ k$ show any barb. In the same way, neither $(a(X)|b(Y) \triangleright R)$ nor $(b(Y)|a(X) \triangleright R)$ show any barb. Since the context is the same, the two processes show the same barbs (in both directions).

If the process $\mathbb{C}[P_1, .., P_n]$ does a reduction it is because the context evolved by itself or one of the hole process reduced by itself or because of the interaction between the context and the hole.

If the context performs a reduction by itself, that is $\mathbb{C}[P_1, .., P_n] \to \mathbb{C}'[P_1, .., P_m]$, with $\to$ being an internal communication or an application or a backward communication or a forward communication, then also $\mathbb{C}[Q_1, .., Q_n] \to \mathbb{C}'[Q_1, .., Q_m]$. Let us note that the number of the holes may change because of the reduction. The same reasoning can be applied if the challenge is done by the right process.

If $\mathbb{C}[P_1, .., (\texttt{KillP}\ l\ h\ k), .., P_n] \rightharpoondown \mathbb{C}[P_1, .., (h(W)|l(Z) \triangleright R), .., P_n]$ (for some R) then also $\mathbb{C}[Q_1, .., (\texttt{KillP}\ h\ l\ k), .., Q_n] \rightharpoondown \mathbb{C}[Q_1, .., (l(Z)|h(W) \triangleright R), .., Q_n]$ and we are still in the same relation. If $\mathbb{C}[P_1, .., a(X)|b(Y) \triangleright R, .., P_n]$ reduces it is because of the presence in the context of two messages of the form, let us say, $a\langle S_1 \rangle$ and $b\langle S_2 \rangle$. If so, we have that $\mathbb{C}[P_1, .., a(X)|b(Y) \triangleright R, .., P_n] \to \mathbb{C}'[P_1, .., R\{^{S_1,S_2}/_{X,Y}\}, .., P_m]$ and on the other side $\mathbb{C}[Q_1, .., b(Y)|a(X) \triangleright R, .., Q_n] \to \mathbb{C}'[Q_1, .., R\{^{S'_1,S'_2}/_{Y,X}\}, .., Q_m]$. Since identity is included in $\mathcal{R}$ (it suffices to consider a 0-ary context), and since $(S_i, S'_i) \in \mathcal{R}$ (since they are subterms) we have that $(R\{^{S_1,S_2}/_{X,Y}\}, R\{^{S_1,S_2}/_{Y,X}\}) \in \mathcal{R}$, and so also $(\mathbb{C}'[P_1, .., R\{^{S_1,S_2}/_{X,Y}\}, .., P_m], \mathbb{C}'[Q_1, .., R\{^{S_1,S_2}/_{Y,X}\}, .., Q_m]) \in \mathcal{R}$, as desired. □

Lemma A.20 *Axiom Ex.P and its instantiation are correct with respect to weak bf barbed bisimulation.*

Proof: Let $S(l_1, l_2, l_3)$ be a process of one of the following forms:

$$S(l_1, l_2, l_3) = (\texttt{KillP}\ l_1\ l_2\ l_3)$$
$$S(l_1, l_2, l_3) = (l_1(Z)|l_2(W) \triangleright l_3\langle (h)\texttt{Par}\ Z\ W\ h \rangle \mid \texttt{Rew}\ l)$$

Let T_l be a either a process of the form $T_l = \texttt{Rew}\ l$ or process $T_l = (l(Z) \triangleright Z\ l)$. Let

$$\begin{aligned} \mathcal{R}' = \{ & (\nu l_1, l_2.\, \texttt{addG}(l_1\langle \llparenthesis P \rrparenthesis \rangle \mid {}_2\langle \llparenthesis Q \rrparenthesis \rangle \mid S(l_1, l_2, l))), \\ & (\nu l_1, l_2.\, \texttt{addG}(l\langle (h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h \rangle \mid T_l))\} \\ \mathcal{R} = \{ & (\mathbb{C}[P_1, \ldots, P_n], \mathbb{C}[Q_1, \ldots, Q_n]) \mid (P_i, Q_i) \in \mathcal{R}' \wedge \\ & \mathbb{C}[P_1, \ldots, P_n], \mathbb{C}[Q_1, \ldots, Q_n]\ well\ formed\} \end{aligned}$$

We now prove that the relation $\mathcal{R}$ is a weak bf barbed bisimulation.

Let us consider barbs. Let us note that process $\nu l_1, l_2.\,\texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l))$ does not show any barb, since addG does not add any barb. On the other side, we have that the process $\nu l_1, l_2.\,\texttt{addG}(l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid T_l)$ shows only a barb on l. If the name l is restricted by the outermost context $\mathbb{C}'$ we are done. Otherwise, we have that:

$$\begin{aligned}&\nu l_1, l_2.\,\texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l)) \hookrightarrow^* \\ &\nu l_1, l_2.\,\texttt{addG}(l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid \texttt{Rew}\ l)\end{aligned}$$

showing a weak barb on l. On both sides barbs shown by the outermost contexts $\mathbb{C}$ are banally matched.

Let us consider the reductions. If $\mathbb{C}[P_1, \ldots, P_n]$ reduces it is because the context reduced by itself or because of the hole processes. The first case is banally matched by the process $\mathbb{C}[Q_1, .., Q_n]$ and vice versa. In the second case we have that the process $\nu l_1, l_2.\,\texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l))$ reduces. We have three cases: the hole reduces by itself, the processes added by addG reduce by itself or both reduce. In the first case, if the reduction is the application of the process $S(l_1, l_2, l)$ this step is mimicked by a zero reduction of the left process. If it is an internal communication on the channels l_1, l_2 then we have that:

$$\begin{aligned}&\nu l_1, l_2.\,\texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid (l_1(W)|l_2(Z) \triangleright l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid \texttt{Rew}\ l)) \rightarrow \\ &\nu l_1, l_2.\,\texttt{addG}(l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid \texttt{Rew}\ l))\end{aligned}$$

and this step is matched by a zero execution of the left one, and we are still in the same relation, since in both sides the process in the i-th hole is the same, we can put it in the context (identity belongs to the relation). The second case is banally matched. In the third case, since the process $\nu l_1, l_2.\,\texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l))$ is well formed then in the context $\texttt{addG}(\_)$ there can be just a $\texttt{Rew}\ l_1$ or/and $\texttt{Rew}\ l_2$ able to interact with the hole. Both cases are similar, so we will consider just the first one. Assume

$$\begin{aligned}&\nu l_1, l_2.\,\texttt{addG}((l_1(Z) \triangleright Z\ l_1) \mid l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l)) \hookrightarrow \\ &\nu l_1, l_2.\,\texttt{addG}(((\!|P|\!)\ l_1 \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l))\end{aligned}$$

We have

$$\begin{aligned}&\nu l_1, l_2.\,\texttt{addG}(l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid T_l) \hookrightarrow^* \\ &\nu l_1, l_2.\,\texttt{addG}(\nu h, k.\,(\!|P|\!)\,h \mid (\!|Q|\!)\,k \mid (\texttt{KillP}\ h\ k\ l))\end{aligned}$$

Using Lemma 5.13

$$\begin{aligned}&\nu l_1, l_2.\, \texttt{addG}(\nu h, k.\, (\!|P|\!)\, h \mid (\!|Q|\!)\, k \mid (\texttt{KillP}\ h\ k\ l)) \hookrightarrow^* \\ &\nu l_1, l_2.\, \texttt{addG}(\nu h, k.\, (\!|P|\!)\, h \mid k\langle(\!|Q|\!)\rangle \mid (\texttt{KillP}\ h\ k\ l))\end{aligned}$$

since all the garbage can be moved to addG and since Q does not contain restrictions thanks to Lemma A.14. Now, using α-conversion and exploiting the fact that addG is closed under α-conversion we get: $\nu l_1, l_2.\, \texttt{addG}(\nu h, k.\, (\!|P|\!)\, l_1 \mid l_2\langle(\!|Q|\!)\rangle \mid (\texttt{KillP}\ l_1\ l_2\ l))$. Since h, k are just used by the $\texttt{addG}(\_)$ context we can rewrite the process as $\nu l_1, l_2.\, \texttt{addG}((\!|P|\!)\, l_1 \mid l_2\langle(\!|Q|\!)\rangle \mid (\texttt{KillP}\ l_1\ l_2\ l))$, where the restriction on k, h has been moved to the context.

Let us consider now reductions of $\mathbb{C}[Q_1, \ldots, Q_n]$. We have three cases: the contexts reduces by itself, the hole reduces by itself or hole and context interacts. First case is banally matched by the process $\mathbb{C}[P_1, \ldots, P_n]$. In the second case, if the reduction is just the application of T_l this step is matched by a zero step execution by $\mathbb{C}[P_1, \ldots, P_n]$, and we are still in the same relation. If T_l is already in its trigger form then we have that

$$\begin{aligned}&\nu l_1, l_2.\, \texttt{addG}(l\langle(h)\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ h\rangle \mid l(Z) \triangleright Z\ l) \rightarrow \\ &\nu l_1, l_2.\, \texttt{addG}(\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ l)\end{aligned}$$

and on the other side we have that

$$\nu l_1, l_2.\, \texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid {}_2\langle(\!|Q|\!)\rangle \mid S(l_1, l_2, l) \hookrightarrow^* \nu l_1, l_2.\, \texttt{addG}(\texttt{Par}\ (\!|P|\!)\ (\!|Q|\!)\ l)$$

and since identity is contained in the relation, we are still in the same relation. In the first case, by hypothesis we have that the considered processes are well formed, implying that (in this particular case) there are no two KillP processes waiting on the same channels. Said otherwise, it is not the case that the context interacts with the hole, and we are done.

□

Lemma A.21 *Axiom Ex.A and its instantiation are correct with respect to weak bf barbed bisimulation.*

Proof: Let $S(l_1, l_2, l_3)$ be either process of the form $S(l_1, l_2, l_3) = (\texttt{KillP}\ l_1\ l_2\ l_3)$ or of the form $S(l_1, l_2, l_3) = (l_1(Z)|l_2(W) \triangleright l_3\langle(h)\texttt{Par}\ Z\ W\ h\rangle \mid \texttt{Rew}\ l)$. Let T_l be either a process of the form $T_l = \texttt{Rew}\ l$ or of the form $T_l = (l(Z) \triangleright Z\ l)$. Let $A((\!|P|\!), l)$ be either a process of the form $A((\!|P|\!), l) = l\langle(\!|P|\!)\rangle$ or of the form $A((\!|P|\!), l) = (\!|P|\!)l$.

Let

$$\begin{aligned}
\mathcal{R}_1 &= \{(\nu l'.\, S(l_1,l_2,l') \mid S(l',l_3,l)), (\nu l'.\, S(l_1,l',l) \mid S(l_2,l_3,l')) \mid l_1,l_2,l,l' \in \mathcal{K}\}\\
\mathcal{R}_2 &= \{(\nu l'.\, l'\langle (h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h\rangle \mid T_{l'} \mid S(l',l_3,l)),\\
&\qquad (\nu l_1,l_2.\, \texttt{addG}(A(\llparenthesis P \rrparenthesis, l_1) \mid A(\llparenthesis Q \rrparenthesis, l_2) \mid \nu l'.\, S(l_1,l',l) \mid S(l_2,l_3,l'))\}\\
\mathcal{R}_3 &= \{(\nu l'.\, ((h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ h)l' \mid T_{l'} \mid S(l',l_3,l)),\\
&\qquad (\nu l_1,l_2.\, \texttt{addG}(A(\llparenthesis P \rrparenthesis, l_1) \mid A(\llparenthesis Q \rrparenthesis, l_2) \mid \nu l'.\, S(l_1,l',l) \mid S(l_2,l_3,l')))\}\\
\mathcal{R}_4 &= \{(\nu l'.\, (\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \rrparenthesis\ l') \mid T_{l'} \mid S(l',l_3,l)),\\
&\qquad (\nu l_1,l_2.\, \texttt{addG}(A(\llparenthesis P \rrparenthesis, l_1) \mid A(\llparenthesis Q \rrparenthesis, l_2) \mid \nu l'.\, S(l_1,l',l) \mid S(l_2,l_3,l')))\}\\
\mathcal{R}_5 &= \{(\nu l', h_1, h_2.\, \llparenthesis P \rrparenthesis h_1 \mid \llparenthesis Q \rrparenthesis h_2 \mid S(h_1,h_2,l') \mid T_{l'} \mid S(l',l_3,l)),\\
&\qquad (\nu l_1,l_2.\, \texttt{addG}(A(\llparenthesis P \rrparenthesis, l_1) \mid A(\llparenthesis Q \rrparenthesis, l_2) \mid \nu l'.\, S(l_1,l',l) \mid S(l_2,l_3,l'))\}\\
\mathcal{R}_6 &= \{(l\langle (h)\texttt{Par}\ \llparenthesis P \mid Q \rrparenthesis\ \llparenthesis R \rrparenthesis\ h\rangle \mid T_l), (T_l \mid l\langle (h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \mid R \rrparenthesis\ l\rangle)\}\\
\mathcal{R}_7 &= \{(((h)\texttt{Par}\ \llparenthesis P \mid Q \rrparenthesis\ \llparenthesis R \rrparenthesis\ h)l), (((h)\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \mid R \rrparenthesis\ h)l)\}\\
\mathcal{R}_8 &= \{(\texttt{Par}\ \llparenthesis P \mid Q \rrparenthesis\ \llparenthesis R \rrparenthesis\ l), (\texttt{Par}\ \llparenthesis P \rrparenthesis\ \llparenthesis Q \mid R \rrparenthesis\ l)\}\\
\mathcal{R}_9 &= \{(\nu l_1,l_2.\, \llparenthesis P \mid Q \rrparenthesis l_1 \mid \llparenthesis R \rrparenthesis l_2 \mid S(l_1,l_2,l)),\\
&\qquad (\nu l_1,l_2.\, \llparenthesis P \rrparenthesis l_1 \mid \llparenthesis Q \mid R \rrparenthesis l_2 \mid S(l_1,l_2,l))\}\\
\mathcal{R}_{10} &= \{(\nu l_1.\, \llparenthesis P \mid Q \rrparenthesis l_1 \mid S(l_1,l_2,l)),\\
&\qquad (\nu l_1,l_3,l_4.\, \llparenthesis P \rrparenthesis l_1 \mid \llparenthesis Q \rrparenthesis l_3 \mid S(l_1,l_4,l) \mid S(l_3,l_2,l_4))\}\\
\mathcal{R}' &= \cup \mathcal{R}_i\\
\mathcal{R} &= \{(\mathbb{C}[P_1,\dots,P_n], \mathbb{C}[Q_1,\dots,Q_n]) \mid (P_i,Q_i) \in \mathcal{R}' \wedge\\
&\qquad \mathbb{C}[P_1,\dots,P_n], \mathbb{C}[Q_1,\dots,Q_n]\ \mathit{wellformed}\}
\end{aligned}$$

then the relation $\mathcal{R}$ is a weak bf barbed bisimulation.

For the sake of brevity we will not consider all the symmetric cases of the various relations $\mathcal{R}_i$ (for example all the cases generated from $\mathcal{R}_1$ where the right process reads two messages on l_2 and l_3 are omitted), and in the bisimulation game we will just consider challenges of the left processes. Let us consider the barbs. All the processes of the form T_l, $S(l_1,l_2,l_3)$, $A(\llparenthesis P \rrparenthesis, l)$ and the $\texttt{addG}(\_)$ do not add any barbs (by Lemma 5.16). Moreover, all the messages on channels $l \in \mathcal{K}$ are on restricted channels. So the only barbs are those shown by the context $\mathbb{C}[\_]$ and by all the processes derived by the application of a translation. In the second case it is easy to verify that these barbs are matched.

Let us now consider reductions. Banally all the reductions performed by the context are matched, and since all the relations are closed under the applications of auxiliary processes such as T_l or $S(l_1, l_2, l_3)$ we will not consider them. Let us start from processes belonging to the relation R_1. If the reduction is an application we remain in the same relation, since $\mathcal{R}_1$ is closed under application. If R_1 reduces because of $S(l_1, l_2, l') = (l_1(Z)|l_2(W) \triangleright l'\langle(h)\texttt{Par}\ Z\ W\ h\rangle)$, this implies that in the context there are two messages on l_1, l_2, let us say $l_1\langle(\!|P|\!)\rangle$, $_2\langle(\!|Q|\!)\rangle$, and with this reduction we move to the relation $\mathcal{R}_2$. From $\mathcal{R}_2$ the (left) process can interact with the context by reading a message on l_3 of the form $l_3\langle(\!|R|\!)\rangle$ and the result of this reduction is contained in $\mathcal{R}_6$, or the process can reduces by itself by reverting the message on the channel l' and this reduction is contained in $\mathcal{R}_3$. In $\mathcal{R}_3$ the only reduction that the left process can do is the application of the Par and with this reduction we move to relation $\mathcal{R}_4$. Also in $\mathcal{R}_4$ the only possible reduction is an application and we move to the relation $\mathcal{R}_5$. In $\mathcal{R}_5$ we have that either $(\!|P|\!)h_1$ or $(\!|Q|\!)h_2$ reduces by means of an application. Let us note that the right process weakly reduces to the left one. In fact since contexts are well formed there exist in the addG two Rew processes, on l_1 and l_2. Hence we have that

$$\begin{aligned}
&\nu l_1, l_2.\, \texttt{addG}(l_1\langle(\!|P|\!)\rangle \mid l_2\langle(\!|Q|\!)\rangle \mid (\texttt{Rew}\ l_1) \mid \\
&\qquad (\texttt{Rew}\ l_2) \mid \nu l'.\, S(l_1, l', l) \mid S(l_2, l_3, l')) \hookrightarrow^* \\
&\nu l_1, l_2.\, \texttt{addG}((\!|P|\!)l_1 \mid (\!|Q|\!)l_2 \mid \nu l'.\, S(l_1, l', l) \mid S(l_2, l_3, l'))
\end{aligned}$$

and by α-converting l_1, l_2 into h_1, h_2 we have $\nu h_1, h_2.\, \texttt{addG}((\!|P|\!)h_1 \mid (\!|Q|\!)h_2 \mid \nu l'.\, S(h_1, l', l) \mid S(h_2, l_3, l'))$ that is equivalent to $\nu h_1, h_2.\, \texttt{addG}((\!|P|\!)h_1 \mid (\!|Q|\!)h_2) \mid \nu l'.\, S(h_1, l', l) \mid S(h_2, l_3, l')$. Since each process R is weakly barbed bf bisimilar to $\texttt{addG}(R)$ (by Proposition 5.1), we conclude by transitivity noting that by ignoring the addG we get back to the relation $\mathcal{R}_1$. From $\mathcal{R}_6$ the only possible reduction is to read from the channel l and this leads us to the relation $\mathcal{R}_7$ (note that this step is banally matched by the right process). Once again, from $\mathcal{R}_7$ the only possible reduction is an application that leads us to the relation $\mathcal{R}_8$. From $\mathcal{R}_8$ we can just move to $\mathcal{R}_9$ by applications. From $\mathcal{R}_9$ there are two possible execution, either the application of $(\!|P \mid Q|\!)l_1$ or the application $(\!|R|\!)l_2$. If the application is the first one we get back to the relation $\mathcal{R}_1$ (by using α-conversion), while if it is the second one we move to the relation $\mathcal{R}_{10}$, by executing in the right process the application $(\!|Q \mid R|\!)l_2$ and then by α-convention we obtain that both the processes (left and right) have a sub-term of the form $(\!|R|\!)l_2$. From $\mathcal{R}_{10}$, the left process can just perform the application on $(\!|P \mid Q|\!)$ and we get back to the relation $\mathcal{R}_1$ (the right process matches this challenge by a 0 steps computation). □

Lemma A.22 *Axiom Ex.Unfold and its instantiation are correct with respect to weak bf barbed bisimulation.*

Proof: Let S be a process of the form $S = \texttt{Rew}\ l$ or $S = (l(Z) \triangleright Z\ l)$. Let $\mathcal{R}' = \{((\!|P|\!)l, \nu\tilde{u}.\, l\langle(\!|Q|\!)\rangle \mid S) \mid P \equiv \nu\tilde{u}.\, Q\} \cup \{((\!|P|\!)l, \nu\tilde{u}.\, (\!|Q|\!)l) \mid P \equiv \nu\tilde{u}.\, Q\}$. Let $\mathcal{R} = \{(\mathbb{C}[P_1, .., P_n]\ ,\ \mathbb{C}[Q_1, .., Q_n]) \mid (P_i, Q_i) \in \mathcal{R}' \wedge i = 1..n \wedge n \in \mathbb{N}\}$. We now show that the relation $\mathcal{R}$ is a weak bf barbed bisimulation.

Let us consider barbs. Since $(\!|P|\!)l$ is an application then the only barbs shown by a process of the form $\mathbb{C}[P_1, .., P_n]$ are those shown by the context. And these barbs are shown also by the process $\mathbb{C}[Q_1, .., Q_n]$. On the other hand, a process of the form $l\langle(\!|Q|\!)\rangle$ shows a barb on l. If this name is restricted by the context we are done, otherwise by using Lemma 5.13 we have that $(\!|P|\!)l \hookrightarrow^* \nu\tilde{v}.\, l\langle(\!|Q|\!)\rangle \mid \texttt{Rew}\ l \mid S$, with S a parallel composition of garbage processes and $l \notin \tilde{v}$. That is $(\!|P|\!)l$ has a weak barb on l.

Let us now consider the reductions. On both sides, reductions done by the context itself are matched. If the reduction is done by a hole process of the form $(\!|P|\!)l$, then we have that the only reduction that can be done is an application. If so, we have to do a case analysis on the form of P. If $P = \nu a.\, P'$, then $\mathbb{C}[P_1, .., (\!|\nu a.\, P'|\!)l, .., P_n] \rightharpoonup^* \mathbb{C}[P_1, .., \nu a.\, (\!|P'|\!)l, .., P_n]$, and this step is mimicked by a zero one of the process $\mathbb{C}[Q_1, .., \nu\tilde{u}.\, l\langle(\!|Q|\!)\rangle, .., Q_n]$,since we can extract name a from the set $\tilde{u}$, as desired. If $P = P' \mid P''$, we have that $P' \mid P'' \equiv \nu\tilde{u}.\, Q$ implies $\nu\tilde{u}.\, Q \equiv \nu\tilde{u}_1.\, Q' \mid \nu\tilde{u}_2.\, Q''$. Hence

$$\begin{array}{l}\mathbb{C}[P_1, .., (\!|P' \mid P''|\!)l, .., P_n] \rightharpoonup \\ \mathbb{C}[P_1, .., \nu l_1, l_2.\, (\!|P'|\!)l_1 \mid (\!|P''|\!)l_2 \mid (\texttt{KillP}\ l_1\ l_2\ l), .., P_n]\end{array}$$

And this step can be matched by

$$\begin{array}{l}\mathbb{C}[Q_1, .., \nu\tilde{u}.\, l\langle(\!|Q' \mid Q''|\!)\rangle \mid \texttt{Rew}\ l, .., Q_n] \rightharpoonup \\ \mathbb{C}[Q_1, .., \nu\tilde{u}.\, l\langle(\!|Q' \mid Q''|\!)\rangle \mid (l(Z) \triangleright Z\ l), .., Q_n] \rightarrow \\ \mathbb{C}[Q_1, .., \nu l_1, l_2.\, \nu\tilde{u}_1.\, (\!|Q'|\!)l_1 \mid \nu\tilde{u}_2.\, (\!|Q''|\!)l_2 \mid (\texttt{KillP}\ l_1\ l_2\ l), .., Q_n]\end{array}$$

and we are still in the same relation since:

$$\begin{array}{c}((\!|P'|\!), \nu\tilde{u}_1.\, (\!|Q'|\!)) \in \mathcal{R}' \\ ((\!|P''|\!), \nu\tilde{u}_2.\, (\!|Q''|\!)) \in \mathcal{R}'\end{array}$$

The other cases are similar.

If the step is performed by the process on the right, we have several cases. If it is the application of a process $\texttt{Rew}\ l$ then this step is matched by a zero reductions of the left process. If the interaction involves the applied form of $(l(Z) \triangleright Z\ l)$ and the message on l we have then $\mathbb{C}[Q_1, .., \nu\tilde{u}.\, l\langle \llparenthesis Q \rrparenthesis \rangle \mid l(Z) \triangleright Z\ l, .., Q_n] \rightarrow \mathbb{C}[Q_1, .., \nu\tilde{u}.\, \llparenthesis Q \rrparenthesis l, .., Q_n]$ and we are still in the same relation. If the reduction is due to $\llparenthesis Q \rrparenthesis l$ then we proceed by case analysis as before and we remain in the same relation, as desired. □

Lemma A.23 *Axiom Ex.Adm and its instantiation are correct with respect to weak bf barbed bisimulation.*

Proof: Let

$$\mathcal{R}' = \{(\nu c.\, (c\langle P \rangle \mid c(Z) \triangleright Z\ k), \llparenthesis P \rrparenthesis k)\}$$
$$\mathcal{R} = \{(\mathbb{C}[P_1, .., P_n], \mathbb{C}[Q_1, .., Q_n]) \mid (P_i, Q_i) \in \mathcal{R}' \wedge i = 1..n \wedge n \in \mathbb{N}\}$$

We now show that the relation $\mathcal{R}$ is a weak bf barbed bisimulation.

Let us consider barbs. We can notice that the name c is restricted so the process $\nu c.\, (c\langle P \rangle \mid (c(Z) \triangleright Z\ k)$ does not show any barb. So if $\mathbb{C}[P_1, .., P_n]$ shows a barb it is because of the context. If so, the same barb is shown by the process $\mathbb{C}[Q_1, .., Q_n]$ too. On the other side, we can notice that since the process $\llparenthesis P \rrparenthesis k$ is an application it does not show any barb. We can use the same reasoning on context barbs as before.

Let us consider now the reductions. If $\mathbb{C}[P_1, .., P_n]$ reduces, it is because one process hole evolved by itself or because the context evolved. In the first case, the only reduction a process hole can do is the communication through the channel c, that is $\mathbb{C}[P_1, .., (\nu c.\, (c\langle P \rangle \mid c(Z) \triangleright Z\ k), .., P_n] \rightarrow \mathbb{C}[P_1, .., \llparenthesis P \rrparenthesis k, .., P_n]$ and this reduction is mimicked by the right process with a zero step reduction. So we have that the left process reduces to the right one, and since identity is included in $\mathcal{R}$ we are done. If the context evolves we have that $\mathbb{C}[P_1, .., P_n] \rightarrow \mathbb{C}'[P_1, .., P_m]$, since the number of holes may change because of the reduction. On the other side we have that, $\mathbb{C}[Q_1, .., Q_n] \rightarrow \mathbb{C}'[Q_1, .., Q_m]$ and we are still in the same relation.

If $\mathbb{C}[Q_1, .., Q_n]$ reduces, it is because a hole processes evolved by itself or because of the context. In the first case we have that the only reduction a process hole can do is an application, that is $\mathbb{C}[Q_1, .., \llparenthesis P \rrparenthesis k, .., Q_n] \rightharpoondown \mathbb{C}[Q_1, .., R, .., Q_n]$, with $(\llparenthesis P \rrparenthesis k) \rightarrow R$. If so, we have that the right process

$$\mathbb{C}[P_1, .., \nu c.\, (c\langle \llparenthesis P \rrparenthesis \rangle \mid c(Z) \triangleright Z\ k, .., P_n)] \rightarrow$$
$$\mathbb{C}[P_1, .., \llparenthesis P \rrparenthesis k, .., P_n)] \rightharpoondown \mathbb{C}[P_1, .., R, .., P_n)]$$

and we still are in the same relation, as desired. If the context evolves, then we can apply the same reasoning of the left process and we are done. □

Now we can use all the previous Lemmas on the single axioms of $\equiv_{Ex}$ to state that $\equiv_{Ex}$ is a weak bf barbed bisimulation.

Proposition 5.2. For any HOπ^+process P, Q the relation $\mathcal{R} = \{(P, Q) \mid P \equiv_{Ex} Q\}$ is a weak bf bisimulation.

Proof: By definition $P \equiv_{Ex} Q$ iff there are $P_1, \dots P_n$ such that $P \equiv_{Ex} P_1 \equiv_{Ex} \dots \equiv_{Ex} P_n \equiv_{Ex} Q$ where each equivalence is obtained by applying just one axiom per time. The proof is by induction on n. The base case is banally verified. In the inductive case we proceed by case analysis on the last applied axiom:

Ex.C : by inductive hypothesis we have that $P \equiv_{Ex} P_n$ implies that $P \overset{\circ}{\approx}_c P_n$ and that $P_n \equiv_{Ex} Q$ using the axiom $Ex.C$. By Lemma A.19 we know that $P_n \equiv_{Ex} Q$ implies $P_n \overset{\circ}{\approx}_c Q$, and by transitivity we have that also $P \overset{\circ}{\approx}_c Q$

Ex.P: by using Lemma A.20 and the same reasoning of the above case.

Ex.A: by using Lemma A.21 and the same reasoning of the first case.

Ex.Unfold: by using Lemma A.22 and the same reasoning of the first case.

Ex.Adm: by using Lemma A.23 and the same reasoning of the first case.

$\equiv_\pi$**:** by using Lemma 5.23 and the same reasoning of the first case.

□