Projet PiCoq

Deliverable D123

December 2012

1 Task 1

Task 1.1 is devoted to finding mathematical frameworks for programming language theory, which would be equipped with (1) mathematical tools to generate them, (2) semantical models, and (3), in a longer-term perspective, notions of morphism which would initiate a general approach to compilation.

Pous and Hirschowitz have proposed a new game model of Milner's Calculus of Communicating Systems (CCS), in which they have defined an analogue of fair testing equivalence (Natarajan and Cleaveland; Brinksma et al.). These results are published in conference and journal versions (see below for the journal version).

Hirschowitz has proved that this semantic fair testing equivalence is fully abstract for standard fair testing equivalence. This work has been rejected from a conference, and will be resubmitted for publication.

What is promising is that a novel algebraic setting has been used as a tool in the proof of full abstraction, which seems very efficient to bridge the gap between syntax and semantics. If this setting could be generalized to more languages (e.g., the -calculus or the -calculus), it might match the main goal of Task 1.1 (2). Current work investigates such generalizations. To date, we have defined an instance of the algebraic framework for the -calculus.

New publications (appended at the end of this report):

- "Innocent Strategies as Presheaves and Interactive Equivalences for CCS", by Tom Hirschowitz and Damien Pous.

2 Task 2

During the last year, Damien Pous developed a new algorithm for language equivalence of finite state automata, together with Filippo Bonchi. This new algorithm exploits well-known ideas from concurrency theory (bisimulations and bisimulations up-to); it happens to be much faster than the other existing algorithms, including the recent ones based on antichains. The correctness of this algorithm has been formalized in Coq.

Alan Schmitt, in collaboration with Ivan Lanese, Michal Lienhardt, Claudio Antares Mezzina, and Jean-Bernard Stefani, designed a higher order reversible process calculus where the reversibility can be finely controlled. They then showed how to encode several existing mechanisms for concurrent reliable systems.

New publications (appended at the end of this report):

- "Checking nfa equivalence with bisimulations up to congruence", by Filippo Bonchi and Damien Pous.
- "Concurrent Flexible Reversibility", by Ivan Lanese, Michal Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani.

3 Task 3

In Task 3, a new perspective on the study of techniques for establishing behavioural equivalences for concurrent systems has emerged in the last year, in conjunction with the beginning of the PhD of Jean-Marie Madiot (co-direction of the PhD with Davide Sangiorgi, Univ. of Bologna).

More precisely, the focus is on typed behavioural equivalences, whereby typing constraints are enforced on the observing contexts. This way, intuitively, more equivalences are valid, since less observations are permitted. A standard type system for behavioural equivalences in process calculi is Pierce and Sangiorgi's system of capability types (also called i/o-types). Types are an important tool to reason about name-passing calculi, and provide expressive proof techniques for that.

We have studied how i/o-types can be enriched, and adapted to a version of the pi-calculus that enjoys symmetry and duality properties. We have demonstrated how the resulting framework makes it possible to relate two rather different encodings of the lambda-calculus in the pi-calculus. Our results rely on the usage of i/o-types, and of typed behavioural equivalences.

New publications (appended at the end of this report):

- "Duality and i/o-Types in the -Calculus", by Daniel Hirschkoff and Jean-Marie Madiot and Davide Sangiorgi.

Scientific Annals of Computer Science vol. 22 (1), 2012, pp. 147–199
DOI: 10.7561/SACS.2012.1.147

Innocent Strategies as Presheaves and Interactive Equivalences for CCS[1]

Tom HIRSCHOWITZ[2], Damien POUS[3]

Abstract

Seeking a general framework for reasoning about and comparing programming languages, we derive a new view of Milner's CCS [34]. We construct a category $\mathbb{E}$ of *plays*, and a subcategory $\mathbb{V}$ of *views*. We argue that presheaves on $\mathbb{V}$ adequately represent *innocent* strategies, in the sense of game semantics [20]. We equip innocent strategies with a simple notion of interaction.

We then prove decomposition results for innocent strategies, and, restricting to presheaves of finite ordinals, prove that innocent strategies are a final coalgebra for a polynomial functor [27] derived from the game. This leads to a translation of CCS with recursive equations.

Finally, we propose a notion of *interactive equivalence* for innocent strategies, which is close in spirit to Beffara's interpretation [1] of testing equivalences [7] in concurrency theory. In this framework, we consider analogues of *fair* testing and *must* testing. We show that must testing is strictly finer in our model than in CCS, since it avoids what we call 'spatial unfairness'. Still, it differs from fair testing, and we show that it coincides with a relaxed form of fair testing.

Keywords: programming language semantics, concurrency, presheaf models, game semantics, behavioural equivalences, fair testing

Note: This is an expanded version of our ICE '11 paper [19]. It notably simplifies a few aspects of the development, and corrects the mistaken statement that fair and must testing coincide in our semantic framework. Must testing only coincides with a relaxed variant of fair testing. This version also subsumes a previous preprint, providing more compact proofs.

[1]Both authors have been partially funded by the French projects CHoCo (ANR-07-BLAN-0324), PiCoq (ANR-10-BLAN-0305-01), and CNRS PEPS CoGIP.

[2]CNRS, Université de Savoie, France. Email: `tom.hirschowitz@univ-savoie.fr`

[3]CNRS, Laboratoire d'Informatique de Grenoble, France.
Email: `damien.pous@ens-lyon.fr`

1 Overview

Theories of programming languages Research in programming languages is mainly technological. Indeed, it heavily relies on techniques which are ubiquitous in the field, but almost never formally made systematic. Typically, the definition of a language then quotiented by variable renaming (α-conversion) appears in many theoretical papers about functional programming languages. Why isn't there yet any abstract framework performing these systematic steps for you? Because the quest for a real *theory* of programming languages is not achieved yet, in the sense of a corpus of results that actually help developing them or reasoning about them. However, many attempts at such a theory do exist.

A problem for most of them is that they do not account for the dynamics of execution, which limits their range of application. This is for example the case of Fiore et al.'s second-order theories [10, 15, 16]. A problem for most of the other theories of programming languages is that they neglect denotational semantics, i.e., they do not provide a notion of model for a given language. This is for example the case of Milner et al.'s *bigraphs* [22], or of most approaches to *structural operational semantics* [37], with the notable exception of the *bialgebraic semantics* of Turi and Plotkin [41]. A recent, related, and promising approach is *Kleene coalgebra*, as advocated by Bonsangue et al. [2]. Finally, *higher-order rewriting* [36], and its semantics in double categories [12] or in cartesian closed 2-categories [18], is not currently known to adequately account for process calculi.

Towards a new approach The most relevant approaches to us are bialgebraic semantics and Kleene coalgebra, since the programme underlying the present paper concerns a possible alternative. A first difference, which is a bit technical but may be of importance, is that both bialgebraic semantics and Kleene coalgebra are based on labelled transition systems (LTSs), while our approach is based on reduction semantics. This seems relevant, since reduction semantics is often considered more primitive than LTSs, and much work has been devoted to deriving the latter from the former [40, 29, 22, 39, 38].

More generally, our approach puts more emphasis on interaction between programs, and hence is less interesting in cases where there is no interaction. A sort of wild hope is that this might lead to unexpected models of programming languages, e.g., physical ones. This could also involve finding a good notion of morphism between languages, and possibly propose a notion of compilation. At any rate, the framework is not set up yet, so

investigating the precise relationship with bialgebraic semantics and Kleene coalgebra is deferred to further work.

How will this new approach look like? Compared to such long-term goals, we only take a small step forward here, by considering a particular case, namely Milner's CCS [34], and providing a new view of it. This view borrows ideas from the following lines of research: game semantics [20], and in particular the notion of an *innocent strategy*, *graphical games* [8, 17], Krivine realisability [28], ludics [13], testing equivalences in concurrency [7, 1], the presheaf approach to concurrency [24, 25], and sheaves [32]. It is also, more remotely, related to graph rewriting [9] and computads [4].

From strategies to presheaves Game semantics [20] has provided fully complete models of programming languages. It is based on the notion of a *strategy*, i.e., a set of *plays* in some game, satisfying a few conditions. In concurrency theory, taking as a semantics the set of accepted plays, or 'traces', is known as *trace semantics*. Trace semantics is generally considered too coarse, since it equates, for a most famous example, the right and the wrong coffee machines, $a.(b+c)$ and $ab+ac$ [34].

An observation essentially due to Joyal, Nielsen, and Winskel is that strategies, i.e., prefix-closed sets of plays, are actually particular *presheaves of booleans* on the category $\mathbb{C}$ with plays as objects, and prefix inclusions as morphisms. By presheaves of booleans on $\mathbb{C}$ we here mean functors $\mathbb{C}^{op} \to 2$, where 2 is the preorder category $0 \leq 1$. If a play p is *accepted*, i.e., mapped to 1, then its prefix inclusions $q \hookrightarrow p$ are mapped to the unique morphism with domain 1, i.e., id_1, which entails that q is also accepted.

Following Joyal, Nielsen, and Winskel, we observe that considering instead presheaves (of sets) on $\mathbb{C}$ yields a much finer semantics. So, a play p is now mapped to a set $S(p)$, to be thought of as the set of ways for p to be accepted by the strategy S. Considering the set of players as a team, $S(p)$ may also be thought of as the set of possible *states* of the team after playing p – which is empty if the team never accepts to play p.

This presheaf semantics is fine enough to account for bisimilarity [24, 25]. Indeed, presheaves are essentially forests with edges labelled by moves. For example, in the setting where plays are finite words on an alphabet, the wrong coffee machine may be represented by the presheaf S defined by the equations on the left and pictured as on the right:

$$
\begin{array}{l}
S(\epsilon) = \{\star\}, \\
S(a) = \{x, x'\}, \\
S(ab) = \{y\}, \\
S(ac) = \{y'\},
\end{array}
\qquad
\begin{array}{l}
S(\epsilon \hookrightarrow a) = \{x \mapsto \star, x' \mapsto \star\}, \\
S(a \hookrightarrow ab) = \{y \mapsto x\}, \\
S(a \hookrightarrow ac) = \{y' \mapsto x'\} :
\end{array}
\qquad
\begin{array}{ccccc}
 & & \overset{a}{\diagup} & \overset{\star}{} & \overset{a}{\diagdown} & \\
 & x & & & & x' \\
{\scriptstyle b}| & & & & & |{\scriptstyle c} \\
 & y & & & & y'.
\end{array}
$$

So, in summary: the standard notion of strategy may be generalised to account for branching equivalences, by passing from presheaves of booleans to presheaves of sets.

Multiple players Traditional game semantics mostly emphasises two-player games. There is an implicit appearance of three-player games in the definition of composition of strategies, and of four-player games in the proof of its associativity, but these games are never given a proper status. A central idea of graphical games, and to a lesser extent of ludics, is the emphasis on multiple-player games.

Here, there first is a base category $\mathbb{B}$ of *positions*, whose objects represent configurations of players. Since the game represents CCS, it should be natural that players are related to each other via the knowledge of *communication channels*. So, roughly, positions are bipartite graphs with vertex sets *players* and *channels*, and edges from channels to players indicating when the former is known to the latter. As a first approximation, morphisms of positions may be thought of as just embeddings of such graphs.

Second, there is a category $\mathbb{E}$ of *plays*, with a functor to $\mathbb{B}$ sending each play to its initial position. Plays are represented in a more flexible way than just sequences of moves, namely using a kind of string diagrams. This echoes the idea [33] that two moves may be independent, and that plays should not depend on the order in which two independent moves are performed. Furthermore, our plays are a rather general notion, allowing, e.g., to focus on a given player. Morphisms of plays account both for:

- prefix inclusion, i.e., inclusion of a play into a longer play, and
- position enlargement, e.g., inclusion of information about some players into information about more players.

Now, restricting to plays over a given initial position X, and then taking presheaves on this category $\mathbb{E}_X$, we have a category of strategies on X.

Innocence A fundamental idea of game semantics is the notion of *innocence*, which says that players have a restricted *view* of the play, and that their actions may only depend on that view.

We implement this here by defining a subcategory $\mathbb{V}_X \hookrightarrow \mathbb{E}_X$ of *views* on X, and deeming a presheaf F on $\mathbb{E}_X$ *innocent* when it is determined by its restriction F' to $\mathbb{V}_X$, in the sense that it is isomorphic to the *right Kan extension* [31] of F' along $\mathbb{V}_X^{op} \hookrightarrow \mathbb{E}_X^{op}$.

We then define *innocent strategies* to be just presheaves on $\mathbb{V}_X$, and view them as (naive) strategies via the (essential) embedding $\widehat{\mathbb{V}_X} \hookrightarrow \widehat{\mathbb{E}_X}$ induced by right Kan extension.

Interaction For each position X, we thus have a category $\mathsf{S}_X = \widehat{\mathbb{V}_X}$ of innocent strategies. In game semantics, composition of strategies is achieved in two steps: *interaction* and *hiding*. Essentially, interaction amounts to considering the three-player game obtained by letting two two-player games interact at a common interface. Hiding then forgets what happens at that interface, to recover a proper two-player game.

We have not yet investigated hiding in our approach, but, thanks to the central status of multiple-player games, interaction is accounted for in a very streamlined way. For any position X with two subpositions $X_1 \hookrightarrow X$ and $X_2 \hookrightarrow X$ such that each player is in either X_1 or X_2, but none is in both, given innocent strategies $F_1 \in \mathsf{S}_{X_1}$ and $F_2 \in \mathsf{S}_{X_2}$, there is a unique innocent strategy, the *amalgamation* $[F_1, F_2]$ of F_1 and F_2, whose restrictions to X_1 and X_2 are F_1 and F_2.

Amalgamation in this sense models interaction in the sense of game semantics, and, using the correspondence with presheaves on $\mathbb{E}_X$ given by right Kan extension, it is the key to defining interactive equivalences.

CCS Next, we define a translation of CCS terms with recursive equations into innocent strategies. This rests on *spatial* and *temporal* decomposition results for innocent strategies. Spatial decomposition says that giving a strategy on a position X is the same as giving a strategy for each of its players. Temporal decomposition says that a strategy is determined up to isomorphism by its set of initial states, plus what remains of each of them after each basic move. Restricting to presheaves of finite ordinals, we also prove that innocent strategies form a final coalgebra for a polynomial functor (in the sense of Kock [27]) derived from the game, thus hinting at links with Kleene coalgebra. It is then easy to translate finite CCS into the language

induced by our polynomial functor, and to finally extend the translation to CCS with recursive equations via infinite unfolding.

A natural question is then: which equivalence does this translation induce on CCS terms? As explained in the following paragraph, we provide some preliminary results about interactive equivalences, but essentially leave the question open.

Interactive equivalences Returning to our model, we then define a notion of *interactive equivalence*, which is close in spirit to both testing equivalences in concurrency theory and Krivine realisability and ludics.

The game, as sketched above, allows interacting with players which are not part of the considered position. E.g., a player in the considered position X may perform an input which is not part of any synchronisation. A *test* for an innocent strategy F on X is then, roughly, an innocent strategy G on a position X' with the same channels as X. To decide whether F *passes* the test G, we consider a restricted variant of the game on the 'union' $X \cup X'$, forbidding any interaction with the outside. We call that variant the *closed-world* game.

Then F passes G iff the amalgamation $[F, G]$, right Kan extended to $\mathbb{E}_{X \cup X'}$ and then restricted to the closed-world game, belongs to some initially fixed class of strategies, $\perp\!\!\!\perp_{X \cup X'}$. Finally, two innocent strategies F and F' on X are equivalent when they pass the same tests.

Here are two examples for $\perp\!\!\!\perp$. Consider a *tick* move, fixed in advance. Then call *successful* all plays containing at least one tick, and accordingly call successful all states reached after a successful play. One may consider:

- $\perp\!\!\!\perp^m$, consisting of strategies whose maximal states (those that admit no strict extensions) are all successful; the tick move plays a rôle analogous to the daimon in ludics: it is the only move which is observable from the outside;
- $\perp\!\!\!\perp^f$, consisting of strategies in which all states on *finite* plays admit a successful extension.

From the classical concurrency theory point of view on behavioural equivalences, the first choice clearly mimicks *must* testing equivalence, while the second mimicks *fair* testing equivalence [35, 3].

Consider the processes Ω and $\Omega|\overline{a}$, where Ω is a process doing infinitely many silent transitions. These processes are intuitively quite different: the latter can do an output on the channel a, while the former cannot. They

are however equated by standard must testing equivalence: the infinite trace provided by Ω may prevent the output prefix from being performed. In fact, must testing equivalence heavily relies on the potential unfairness of the scheduler. In the literature, this peculiar behaviour actually motivates the introduction of fair testing equivalence.

In contrast, our notion of play is more flexible than standard traces, so that our counterpart to must testing equivalence actually distinguishes these two processes: the infinite play where the output prefix is not performed is not maximal, so that the corresponding unfair behaviour is not taken into account. In other words, thanks to our notion of play, the rather natural notion of must testing already avoids what we call 'spatial unfairness'. However, must testing does not coincide with fair testing in our setting, because there are other sources of unfairness, that are not handled properly. Technically, we prove that $\perp\!\!\!\perp^m$ coincides with the set of strategies whose states all admit a successful extension. However, the restriction to finite plays in the definition of $\perp\!\!\!\perp^f$ is required to rule out other sources of unfairness.

Summary In summary, our approach emphasises a flexible notion of multiple-player play, encompassing both views in the sense of game semantics, closed-world plays, and intermediate notions. Strategies are then described as presheaves on plays, while innocent strategies are presheaves on views. Innocent strategies admit a notion of interaction, or amalgamation, and are embedded into strategies via right Kan extension. This allows a notion of testing, or interactive equivalence by amalgamation with the test, right Kan extension, and finally restriction to closed-world.

Our main technical contributions are then a translation of CCS terms with recursive equations into innocent strategies, and the study of fair and must equivalences in our setting.

Perspectives Our next task is clearly to tighten the link with CCS. Namely, we should explore which equivalence on CCS is induced via our translation, for a given interactive equivalence. We will start with $\perp\!\!\!\perp^f$. Furthermore, the very notion of interactive equivalence might deserve closer consideration. Its current form is rather *ad hoc*, and one could hope to see it emerge more naturally from the game. For instance, the fixed class $\perp\!\!\!\perp$ of 'successful' strategies should probably be more constrained than is done here. Also, the paradigm of observing via the set of successful tests might admit sensible refinements, e.g., probabilistic ones.

Another possible research direction is to tighten the link with 'graphical' approaches to rewriting, such as graph rewriting or computads. E.g., our plays might be presented by a computad [14], or be the bicategory of rewrite sequences up to shift equivalence, generated by a graph grammar in the sense of Gadducci et al. [11]. Both goals might require some technical adjustments, however. For computads, we would need the usual yoga of U-turns to flexibly model our positions; e.g., zigzags of U-turns are usually only equal up to a higher-dimensional cell, while they would map to equal positions in our setting. For graph rewriting, the problem is that our positions are not exactly graphs (e.g., the channels known to a player are linearly ordered).

Other perspectives include the treatment of more complicated calculi like π or λ. In particular, calculi with duplication of terms will pose a serious challenge. An even longer-term hope is to be able to abstract over our approach. Is it possible to systematise the process starting from a calculus as studied in programming language theory, and generating its strategies modulo interactive equivalence? If this is ever understood, the next question is: when does a translation between two such calculi preserve a given interactive equivalence? Finding general criteria for this might have useful implications in programming languages, especially compilation.

Notation Throughout the paper, we abusively identify n with $\{1 \dots n\}$, for readability. So, e.g., $i \in n$ means $i \in \{1, \dots, n\}$.

The various categories and functors constructed in the development are summed up with a short description in Table 1. There, given two functors $\mathbb{C} \xrightarrow{F} \mathbb{E} \xleftarrow{G} \mathbb{D}$, we denote (slightly abusively) by $\mathbb{C} \downarrow_{\mathbb{E}} \mathbb{D}$ the *comma* category: it has as objects triples (C, D, u) with $C \in \mathbb{C}$, $D \in \mathbb{D}$, and $u \colon F(C) \to G(D)$ in $\mathbb{E}$, and as morphisms $(C, D, u) \to (C', D', u')$ pairs (f, g) making the square above commute. Also, when F is the identity on $\mathbb{C}$ and $G \colon 1 \to \mathbb{C}$ is an object C of $\mathbb{C}$, this yields the usual *slice* category, which we abbreviate as $\mathbb{C}/C$. Finally, the category of presheaves on any category $\mathbb{C}$ is denoted by $\widehat{\mathbb{C}} = [\mathbb{C}^{op}, \mathsf{Set}]$.

$$\begin{array}{ccc} FC & \xrightarrow{F(f)} & FC' \\ {\scriptstyle u}\downarrow & & \downarrow{\scriptstyle u'} \\ GD & \xrightarrow[G(g)]{} & GD' \end{array}$$

We denote by $\mathrm{ob}(\mathbb{C})$ the set of objects of any small category $\mathbb{C}$. For any functor $F \colon \mathbb{C} \to \mathbb{D}$, we denote by $\mathbb{F}^{op} \colon \mathbb{C}^{op} \to \mathbb{D}^{op}$ the functor induced on opposite categories, defined exactly as F on both objects and morphisms. Also, recall that an *embedding* of categories is an injective-on-objects, faithful functor. This admits the following generalisation: a functor $F \colon \mathbb{C} \to \mathbb{D}$ is *essentially injective on objects* when $FC \cong FC'$ implies $C \cong C'$. Any faithful,

Category	Description of its objects
$\widehat{\mathbb{C}}$	'diagrams'
$\mathbb{B} \hookrightarrow \widehat{\mathbb{C}}$	positions
$\mathbb{E} \hookrightarrow (\mathbb{B} \downarrow_{\widehat{\mathbb{C}}} \widehat{\mathbb{C}})$	plays
$\mathbb{E}_X = (\mathbb{E} \downarrow_{\mathbb{B}} (\mathbb{B}/X))$	plays on a position X
$\mathbb{V}_X \hookrightarrow \mathbb{E}_X$	views on X
$\mathsf{S}_X = \widehat{\mathbb{V}_X}$	innocent strategies on X
$\mathbb{W} \hookrightarrow \mathbb{E}$	closed-world plays
$\mathbb{W}(X)$	closed-world plays on X

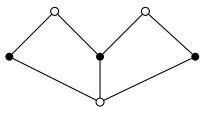
Table 1: Summary of categories and functors

essentially injective on objects functor is called an *essential embedding*.

2 Plays as String Diagrams

We now describe our approach more precisely, starting with the category of multiple-player plays. For the sake of clarity, we first describe this category in an informal way, before giving the precise definition (Section 3).

2.1 Positions



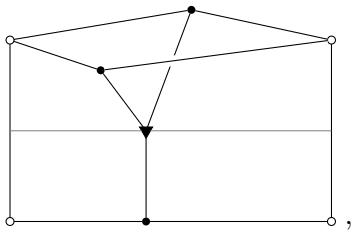
Since the game represents CCS, it should be natural that players are related to each other via the knowledge of *communication channels*. This is represented by a kind of[4] finite, bipartite graph: an example position is on the right. Bullets represent players, circles represent channels, and edges indicate when a player knows a channel. The channels known by a player are linearly ordered. Formally, as explained in Section 3, positions are presheaves over a certain category $\mathbb{C}_1$. Morphisms of positions are natural transformations, which are roughly morphisms of graphs, mapping players to players and channels to channels. In full generality, morphisms thus do not have to be injective, but include in particular embeddings of positions in the intuitive sense. Positions and morphisms between them form a category $\mathbb{B}$.

[4]Only 'a kind of', because, as mentioned above, the channels known to a player are linearly ordered.

2.2 Moves

Plays will be defined as glueings of *moves* between positions. Moves are derived from the very definition of CCS, as we now sketch. The diagrams we draw in this section will be given a very precise combinatorial definition in Section 3.

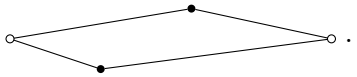
Let us start with the *forking* move, which corresponds to parallel composition in CCS: a process (the player) forks into two sub-processes. In the case of a player knowing two channels, the forking move is represented by the diagram



(1)

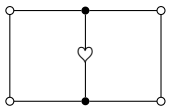
to be thought of as a move from the bottom position X

(with one player p) to the top position Y

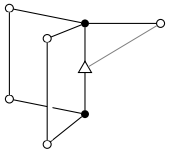


(with two players, which we call the 'avatars' of p). The left- and right-hand borders are just channels evolving in time, not noticing that the represented player forks into two. The surfaces spread between those vertical lines represent links (edges in the involved positions) evolving in time. For example, each link here divides into two when the player forks, thus representing the fact that both of the avatars retain knowledge of the corresponding channel. There is of course an instance π_n of forking for each n, according to the number of channels known to the player. As for channels known to a player, the players and channels touching the black triangle are ordered: there are different 'ports' for the initial player and its two avatars.

We then have a *tick* move $\heartsuit_n$, whose role is to define successful plays, and a move for the *channel creation* or *restriction* of CCS, here ν_n. In the case where the player knows two channels, they are graphically represented as

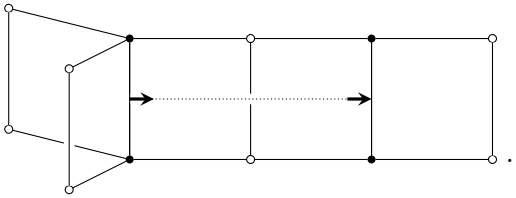


and



respectively. As expected, there is an instance of each of these two moves for each number n of channels known to the player.

We also need a move to model CCS-like synchronisation, between two players. For all n and m, representing the numbers of channels known to the players involved in the synchronisation, and for all $i \in n$, $j \in m$, there is a *synchronisation* $\tau_{n,i,m,j}$, represented, in the case where one player outputs on channel $3 \in 3$ and the other inputs on channel $1 \in 2$, by



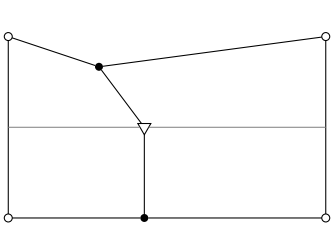
.

As we shall see in Section 3, the dotted wire in the picture is actually a point in the formal representation (i.e., an element of the corresponding presheaf).

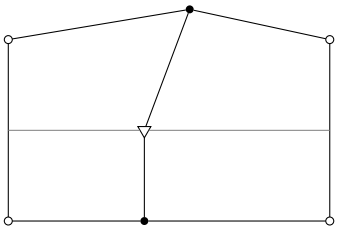
The above four kinds of moves (forking, tick, channel creation, and synchronisation) come from the reduction semantics of CCS. We classify these as *closed-world* moves, since they correspond to the evolution of a group of players in isolation.

We however need a more fine-grained structure for moves: moves whose final position has more than one player (forking and synchronisation) must be decomposed into *basic* moves, to get an appropriate notion of view.

We introduce two sub-moves for forking: *left* and *right half-forking*. In the case where the player knows two channels, they are represented by the following diagrams, respectively:



and



. (2)

These sub-moves represent what each of the 'avatars' of the forking player sees of the move. We call π^l_n and π^r_n the respective instances of the left-hand and right-hand basic moves for a player knowing n channels. Formally, there will be injections from the left and right half-forking moves to the corresponding forking moves.

We finally decompose synchronisation into an input move and an output move: $a.P$ and $\overline{a}.P$ in CCS become $\iota^+_{n,i}$ and $\iota^-_{n,i}$ here (where n is the number of known channels, $i \in \{1 \dots n\}$ is the index of the channel bearing the synchronisation). Here, output on the right-hand channel and input on the left-hand channel respectively look like

and . (3)

Like with forking, there will be injections from the input and output moves to the corresponding synchronisation moves.

All in all, there are three classes of moves, which we summarise in Table 2.

- Tick, channel creation, half-forking, and input/output moves are *basic* moves: they evolve from a position with exactly one player to another position with exactly one player. These moves are used to define views later on.
- Forking, synchronisation, tick and channel creation moves are *closed-world* moves: they correspond to the case where a group of players evolves on its own, in isolation; they are central to the notion of interactive equivalence.
- We need a third class of moves, called *full*, which consists of forking, input, output, tick and channel creation. They involve a single player and all of its avatars. They appear, e.g., in the statement of Lemma 12, which is a partial correctness criterion for closed-world plays.

Formally, we define moves as cospans $X \hookrightarrow P \hookleftarrow Y$ in the category of diagrams (technically a presheaf category $\widehat{\mathbb{C}}$—see Section 3), where X is the initial position and Y the final one. Both legs of the cospan are actually monic morphisms in $\widehat{\mathbb{C}}$, as will be the case for all cospans considered here.

Basic	Full	Closed-world
Left half-forking Right half-forking	Forking	Forking
Input Output	Input Output	Synchronisation
Channel creation	Channel creation	Channel creation
Tick	Tick	Tick

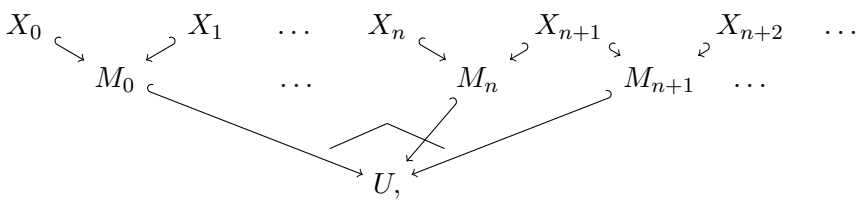
Table 2: Summary of classes of moves

2.3 Plays

We now sketch how plays are defined as glueings of moves. We start with the following example, depicted in Figure 1. The initial position consists of two players p_1 and p_2 sharing knowledge of a channel a, each of them knowing another channel, resp. a_1 and a_2. The play consists of four moves: first p_1 forks into $p_{1,1}$ and $p_{1,2}$, then p_2 forks into $p_{2,1}$ and $p_{2,2}$, and then $p_{1,1}$ does a left half-fork into $p_{1,1,1}$; finally $p_{1,1,1}$ synchronises (as the sender) with $p_{2,1}$. Now, we reach the limits of the graphical representation, but the order in which p_1 and p_2 fork is irrelevant: if p_2 forks before p_1, we obtain the same play. This means that glueing the various parts of the picture in Figure 1 in different orders formally yields the same result (although there are subtle issues in representing this result graphically in a canonical way).

Let us now sketch a definition of plays. Recall that moves may be seen as cospans $X \hookrightarrow M \hookleftarrow Y$, and consider an *extended* notion of move, which may occur in a position not limited to players involved in the move. For example, the moves in Figure 1 are extended moves in this sense.

Definition 1 *A* play *is an embedding $X_0 \hookrightarrow U$ in the category $\widehat{\mathbb{C}}$ of diagrams, isomorphic to a possibly denumerable 'composition' of moves in the (bi)category $\mathsf{Cospan}(\widehat{\mathbb{C}})$ of cospans in $\widehat{\mathbb{C}}$, i.e., obtained as a colimit:*



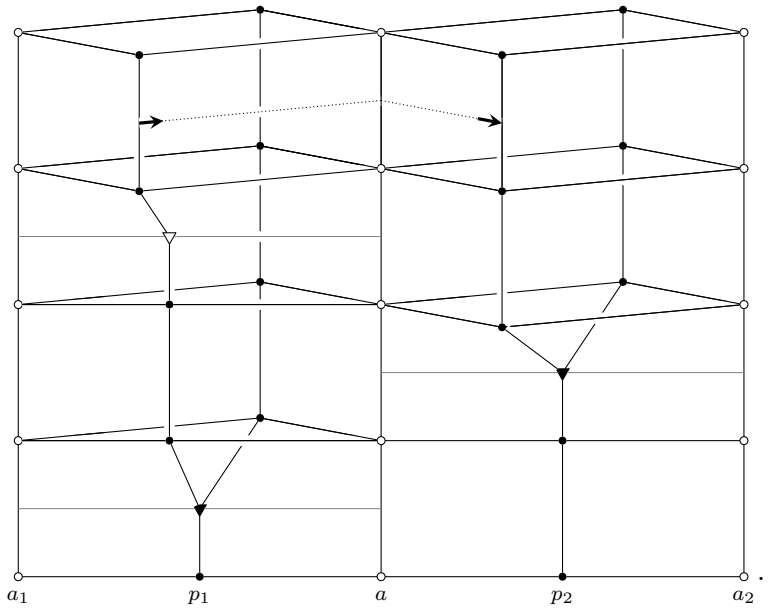


Figure 1: An example play

where each $X_i \hookrightarrow M_i \hookleftarrow X_{i+1}$ *is an extended move.*

We often denote plays just by U, leaving the embedding $X \hookrightarrow U$ implicit.

Remark 1 *For finite plays, one might want to keep track not only of the initial position, but also of the final position. This indeed makes sense. Finite plays then compose 'vertically', and form a double category. But infinite plays do not really have any final position, which explains our definition.*

Let a morphism $(X \hookrightarrow U) \to (Y \hookrightarrow V)$ of plays be a pair (h, k) making the diagram on the right commute in $\widehat{\mathbb{C}}$. This permits both inclusion 'in width' and 'in height'. E.g., the play consisting of the left-hand basic move in (2) embeds in exactly two ways into the play of Figure 1. (Only two because the image of the base position must lie in the base position of the codomain.) We have:

$$\begin{array}{ccc} U & \xrightarrow{k} & V \\ \uparrow & & \uparrow \\ X & \xrightarrow{h} & Y. \end{array}$$

Proposition 1 *Plays and morphisms between them form a category* $\mathbb{E}$.

There is a projection functor $\mathbb{E} \to \mathbb{B}$ mapping each play $X \hookrightarrow U$ to its base position X. This functor has a section, which is an embedding $\mathbb{B} \hookrightarrow \mathbb{E}$, mapping each position X to the 'identity' play $X \hookrightarrow X$ on X.

Remark 2 (Size) *The category $\mathbb{E}$ is only locally small. Since presheaves on a locally small category are less well-behaved than on a small category, we will actually consider a skeleton of $\mathbb{E}$. Because $\mathbb{E}$ consists only of denumerable presheaves, this skeleton is a small category. Thus, our presheaves in the next section may be understood as taken on a small category.*

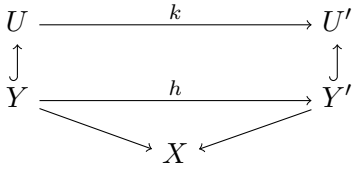
Remark 3 *Plays are not very far from being just (infinite) abstract syntax trees (or forests) 'glued together along channels'. E.g., the play from Figure 1 is the glueing of, say $(\pi_2^l(\overline{a}.0))|0$ and $a|0$ along a.*

2.4 Relativisation

If we now want to restrict to plays over a given base position X, we may consider

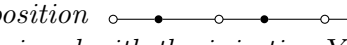
Definition 2 *Let the category $\mathbb{E}_X$ have*

- *as objects pairs of a play $Y \hookrightarrow U$ and a morphism $Y \to X$,*
- *as morphisms $(Y \hookrightarrow U) \to (Y' \hookrightarrow U')$ all pairs (h, k) making the diagram*



commute in $\widehat{\mathbb{C}}$.

We will usually abbreviate $U \hookleftarrow Y \to X$ as just U when no ambiguity arises. As for morphisms of positions, in full generality, h and k, as well as the morphisms $Y \to X$, do not have to be injective.

Example 1 *Let X be the position*  *. The play in Figure 1, say $Y \hookrightarrow U$, equipped with the injection $Y \hookrightarrow X$ mapping the two players of Y to the two leftmost players of X, is an object of $\mathbb{E}_X$.*

One naively could imagine that the objects $\mathbb{E}_X$ could just consist of plays $X \hookrightarrow U$ on X. However, spatial decomposition, Theorem 1, relies on our slightly more complex definition. E.g., still in Figure 1, this allows us to distinguish between the identity view $[2] = [2] \overset{p_1}{\hookrightarrow} X$ on p_1 from the identity view $[2] = [2] \overset{p_2}{\hookrightarrow} X$ on p_2, which would otherwise not be possible.

3 Diagrams

In this section, we define the category on which the string diagrams of the previous section are presheaves. The techniques used here date back at least to Carboni and Johnstone [5, 6].

3.1 First Steps

Let us first consider two small examples. It is well-known that directed graphs form a presheaf category: consider the category $\mathbb{C}$ freely generated by the graph with two vertices, say $\star$ and $[1]$, and two edges $d, c \colon \star \to [1]$ between them. One way to visualise this is to compute the *category of elements* of a few presheaves on $\mathbb{C}$. Recall that the category of elements of a presheaf F on $\mathbb{C}$ is the comma category $y \downarrow_{\widehat{\mathbb{C}}} F$, where y is the Yoneda embedding. Via Yoneda, it has as elements pairs (C, x) with $C \in \mathrm{ob}(\mathbb{C})$ and $x \in F(C)$, and morphisms $(C, x) \to (D, y)$ morphisms $f \colon C \to D$ in $\mathbb{C}$ such that $F(f)(y) = x$ (which we abbreviate as $y \cdot f = x$ when the context is clear).

Example 2 *Consider the graph*

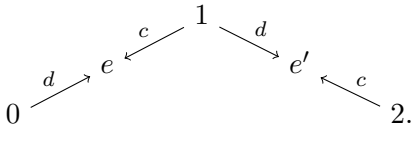
$$0 \xrightarrow{\ e\ } 1 \xrightarrow{\ e'\ } 2$$

with three vertices $0, 1,$ and $2,$ and two edges e and e'.

This graph is represented by the presheaf F defined by the following equations, whose category of elements is actually freely generated by the graph on the right:

- $F(\star) = \{0, 1, 2\}$,
- $F([1]) = \{e, e'\}$,

- $e \cdot d = 0$,
- $e \cdot c = 1$,
- $e' \cdot d = 1$,
- $e' \cdot c = 2$,



This latter graph is not exactly the original one, but it does represent it. Indeed, for each vertex we know whether it is in $F(\star)$ or $F([1])$, hence whether it represents a 'vertex' or an 'edge'. The arrows all go from a 'vertex' v to an 'edge' e. They lie over d when v is the domain of e, and over c when v is the codomain of e.

Multigraphs, i.e., graphs whose edges have a list of sources instead of just one, may also be seen as a presheaves on the category freely generated by the graph with

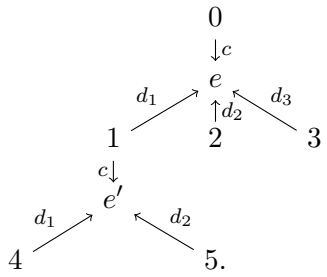
- as vertices: one special vertex $\star$, plus for each natural number n a vertex, say, $[n]$; and
- for all $n \in \mathbb{N}$, $n+1$ edges $\star \to [n]$, called $d_1, \ldots, d_n$, and c.

It should be natural for presheaves on this category to look like multigraphs: the elements of a presheaf F over $\star$ are the vertices in the multigraph, the elements over $[n]$ are the n-ary multiedges, and the action of the d_i's give the ith source of a multiedge, while the action of c gives its target.

Example 3 *Similarly, computing a few categories of elements might help visualising. As above, consider F defined by*

- $F(\star) = \{0, 1, 2, 3, 4, 5\}$,
- $F([1]) = F([0]) = \emptyset$,
- $F([2]) = \{e'\}$,
- $F([3]) = \{e\}$,
- $F([n+4]) = \emptyset$,
- $e \cdot c = 0$,
- $e \cdot d_1 = 1$,
- $e \cdot d_2 = 2$,
- $e \cdot d_3 = 3$,
- $e' \cdot c = 1$,
- $e' \cdot d_1 = 4$,
- $e' \cdot d_2 = 5$,

whose category of elements is freely generated by the graph:



Now, this pattern may be extended to higher dimensions. Consider for example extending the previous base graph with a vertex $[m_1, \dots, m_n; p]$ for all natural numbers $n, p, m_1, \dots, m_n$, plus edges

$$\begin{array}{l} s_1 \colon [m_1] \to [m_1, \dots, m_n; p], \\ \dots, \\ s_n \colon [m_n] \to [m_1, \dots, m_n; p], \mathit{and} \\ t \colon [p] \to [m_1, \dots, m_n; p]. \end{array}$$

Let now $\mathbb{C}$ be the free category on this extended graph. Presheaves on $\mathbb{C}$ are a kind of 2-multigraphs: they have vertices, multiedges, and multiedges between multiedges.

We could continue this in higher dimensions.

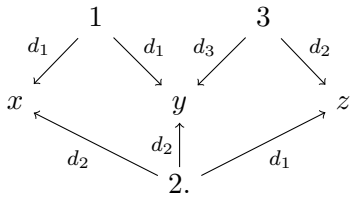
3.2 Constructing the Base Category

Our base category follows a very similar pattern. We start from a slightly different graph: let $\mathbb{G}_0$ have just one vertex $\star$; let $\mathbb{G}_1$, have one vertex $\star$, plus a vertex $[n]$ for each natural number n, plus n edges $d_1, \dots, d_n \colon \star \to [n]$. Let $\mathbb{C}_0$ and $\mathbb{C}_1$ be the categories freely generated by $\mathbb{G}_0$ and $\mathbb{G}_1$, respectively. So, presheaves on $\mathbb{C}_1$ are a kind of hypergraphs with arity (since vertices incident to a hyperedge are numbered). This is enough to model positions.

Example 4 *The position drawn at the beginning of Section 2.1 may be represented as the presheaf*

- $\star \mapsto \{1, 2, 3\}$,
- $[2] \mapsto \{x, z\}$,
- $[3] \mapsto \{y\}$,
- $\_ \mapsto \emptyset$,
- $x \cdot d_1 = 1$,
- $x \cdot d_2 = 2$,
- $z \cdot d_1 = 2$,
- $z \cdot d_2 = 3$,
- $y \cdot d_1 = 1$,
- $y \cdot d_2 = 2$,
- $y \cdot d_3 = 3$,

whose category of elements is:



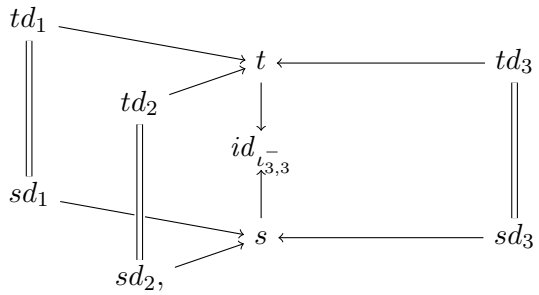
Now, consider the graph $\mathbb{G}_2$, which is $\mathbb{G}_1$ augmented with:

- for all n, vertices $\heartsuit_n$, π^l_n, π^r_n, ν_n,
- for all n and $1 \leq i \leq n$, vertices $\iota^+_{n,i}$ and $\iota^-_{n,i}$,
- for all n, edges $s, t \colon [n] \to \heartsuit_n$, $s, t \colon [n] \to \pi^l_n$, $s, t \colon [n] \to \pi^r_n$, $s \colon [n] \to \nu_n$, $t \colon [n+1] \to \nu_n$,
- for all n and $1 \leq i \leq n$, edges $s, t \colon [n] \to \iota^+_{n,i}$, $s, t \colon [n] \to \iota^-_{n,i}$.

We slightly abuse language here by calling all these t's and s's the same. We could label them with their codomain, but we refrain from doing so for the sake of readability.

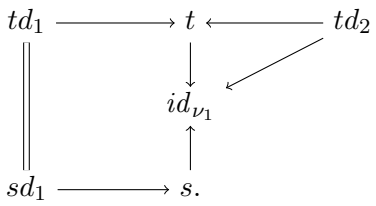
Now, let $\mathbb{C}_2$ be the category generated by $\mathbb{G}_2$ and the relations $s \circ d_i = t \circ d_i$ for all n and $1 \leq i \leq n$ (for all sensible—common—codomains). The intuition here is that for any basic move by a player with n channels, these n channels remain the same after the move. This includes the case of ν_n, for which the absence of any equation involving the new channel makes it different from the others.

Example 5 *Again, computing a few categories of elements is in order. For example, the category of elements of (the representable presheaf on) $\iota^-_{3,3}$ is the poset freely generated by the graph*



to be compared with the corresponding pictures (3).

Example 6 *Similarly, the category of elements of ν_1 is the poset freely generated by the graph*



Note that only channel creation changes the number of channels known to the player, and accordingly the corresponding morphism t has domain $[n+1]$.

Presheaves on $\mathbb{C}_2$ are enough to model basic moves, but since we want more, we continue, as follows.

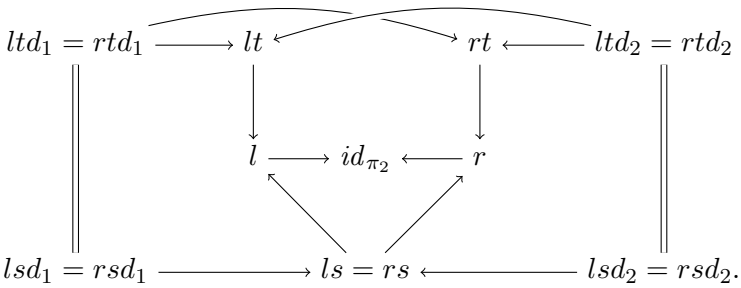
Let $\mathbb{G}_3$ be $\mathbb{G}_2$, augmented with:

- for all n, a vertex π_n, and
- edges $l\colon \pi_n^l \to \pi_n$ and $r\colon \pi_n^r \to \pi_n$.

Definition 3 *Let $\mathbb{C}_3$ be the category generated by $\mathbb{G}_3$, the previous relations, plus the relations $l \circ s = r \circ s$.*

The equation models the fact that a forking move should be played by just one player. We also call $s = l \circ s = r \circ s$ the common composite, which gives a uniform notation for the initial player of full moves.

Example 7 *The category of elements of π_2 is the poset freely generated by the graph*



The two views corresponding to left and right half-forking are subcategories, and the object id_{π_2} 'ties them together'.

Presheaves on $\mathbb{C}_3$ are enough to model full moves; to model closed-world moves, and in particular synchronisation, we continue as follows.

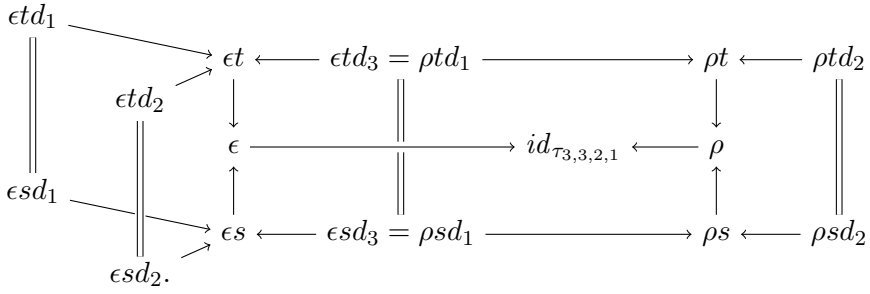
Let $\mathbb{G}_4$ be $\mathbb{G}_3$, augmented with, for all n, m, $1 \leq i \leq n$, and $1 \leq j \leq m$,

- a vertex $\tau_{n,i,m,j}$, and
- edges $\epsilon\colon \iota^+_{n,i} \to \tau_{n,i,m,j}$ and $\rho\colon \iota^-_{m,j} \to \tau_{n,i,m,j}$ (ϵ and ρ respectively stand for 'emission' and 'reception').

Definition 4 *Let $\mathbb{C}_4$ be the category generated by $\mathbb{G}_4$, the previous relations, plus, for each $\iota^+_{n,i} \xrightarrow{\epsilon} \tau_{n,i,m,j} \xleftarrow{\rho} \iota^-_{m,j}$, the relation $\epsilon \circ s \circ d_i = \rho \circ s \circ d_j$.*

This equation is the exact point where we enforce that a synchronisation involves an input and an output on the same channel, as announced in Example 5.

Example 8 *The category of elements of $\tau_{3,3,1,1}$ is the preorder freely generated by the graph*



Again, the two views corresponding to $\iota^+_{3,3}$ and $\iota^-_{2,1}$ are subcategories, and the new object $\tau_{3,3,2,1}$ ties them together.

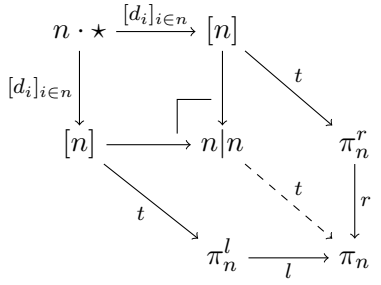
3.3 Positions and Moves

We have now defined the base category $\mathbb{C} = \mathbb{C}_4$ on which the string diagrams of Section 2 are presheaves. More accurately we have defined a sequence $\mathbb{C}_0 \hookrightarrow \ldots \hookrightarrow \mathbb{C}_4$ of subcategories.

Positions Positions are finite presheaves on $\mathbb{C}_1$, or equivalently, finite presheaves on $\mathbb{C}_4$ empty except over $\mathbb{C}_1$.

Moves Basic moves should be essentially representable presheaves on objects in $\mathrm{ob}(\mathbb{C}_2) \setminus \mathrm{ob}(\mathbb{C}_1)$. Recall however that basic moves are defined as particular cospans in $\widehat{\mathbb{C}}$. This is also easy: in the generating graph $\mathbb{G}_2$, each such object c has exactly two morphisms s and t into it, from objects, say, $[n_s]$ and $[n_t]$, respectively. By Yoneda, these induce a cospan $[n_s] \xrightarrow{s} c \xleftarrow{t} [n_t]$ in $\widehat{\mathbb{C}}$, which is the desired cospan. (Observe, again, that only ν_n has $n_s \neq n_t$.)

Similarly, full moves either are basic moves, or are essentially representable presheaves on objects in $\mathrm{ob}(\mathbb{C}_3) \setminus \mathrm{ob}(\mathbb{C}_1)$, i.e., representables on some π_n. To define the expected cospan, first observe that by the equation $ls = rs$, we obtain an morphism $[n] \xrightarrow{s} \pi_n^l \xrightarrow{l} \pi_n$, equal to rs, in $\widehat{\mathbb{C}}$. This will form the first leg of the cospan. For the other, observe that for each n and $i \in n$, we obtain, by the equations $ltd_i = lsd_i = rsd_i = rtd_i$ and by Yoneda, that the outermost part of



(4)

commutes in $\widehat{\mathbb{C}}$, where $n \cdot \star$ denotes an n-fold coproduct of $\star$. Letting $n|n$ be the induced pushout, and the dashed morphism t be obtained by its universal property, we obtain the desired cospan $[n] \xrightarrow{ls} \pi_n \xleftarrow{t} n|n$.

Finally, closed-world moves either are full moves, or are essentially representable presheaves on some $\tau_{n,i,m,j}$. To define the expected cospan, we proceed as in Figure 2: compute the pushout $n\,{}_i|_j\, m$, and infer the dashed morphisms s' and t' to obtain the desired cospan $n_i|_j m \xrightarrow{s'} \tau_{n,i,m,j} \xleftarrow{t'} n_i|_j m$.

Remark 4 (Isomorphisms) *Moves are particular cospans in $\widehat{\mathbb{C}}$. For certain moves, the involved objects are representable, but not for others, like forking or synchronisation, whose final position is not representable. In the latter cases, our definition thus relies on a choice, e.g., of pushout in* (4)*. Thus, let us be completely accurate: a move is a cospan which is isomorphic to one of the cospans chosen above, in $\widehat{\mathbb{C}}^{\cdot\leftarrow\cdot\rightarrow\cdot}$, i.e., the category of functors from the category $\cdot \leftarrow \cdot \rightarrow \cdot$ (generated by the graph with three objects and an edge from one to each of the other two) to $\widehat{\mathbb{C}}$.*

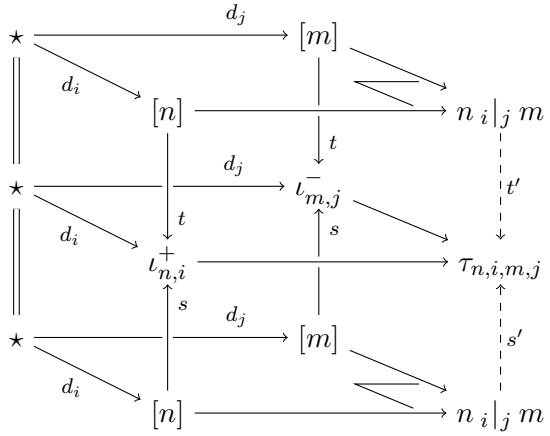


Figure 2: Construction of the synchronisation move

3.4 Extended Moves, Plays, and Relativisation

The most delicate part of our formalisation of Section 3 is perhaps the passage from moves to extended moves. Recall from the paragraph above Definition 1 that an extended move should be like a move occurring in a larger position.

Moves with interfaces To formalise this idea, we first equip moves with interfaces, as standard in graph rewriting [23]. Since moves are cospans, one might expect that interfaces are cospans too. This may be done, but there is a simpler, equivalent presentation. The route we follow here might have to be generalised in order to handle more complex calculi than CCS, but let us save the complications for later work.

Here, we define an *interface* for a cospan $X \to M \leftarrow Y$ to consist of a presheaf I and morphisms $X \leftarrow I \to Y$ such that

$$\begin{array}{ccc} I & \longrightarrow & Y \\ \downarrow & & \downarrow \\ X & \longrightarrow & M \end{array} \tag{5}$$

commutes, and I has dimension 0, i.e., is empty except over $\mathbb{C}_0$, i.e., consists only of channels.

Definition 5 *A cospan equipped with an interface is called a* cospan with interface.

Moves are particular cospans, and we now equip them with canonical interfaces: all moves except channel creation preserve the set of channels, the interface is then $n \cdot \star$, with the obvious inclusion. For example, the less obvious case is π_n: we choose

$$\begin{array}{ccc} n \cdot \star & \longrightarrow & n|n \\ \downarrow & & \downarrow \\ [n] & \longrightarrow & \pi_n, \end{array}$$

where the upper map is as in (4). For channel creation, we naturally choose

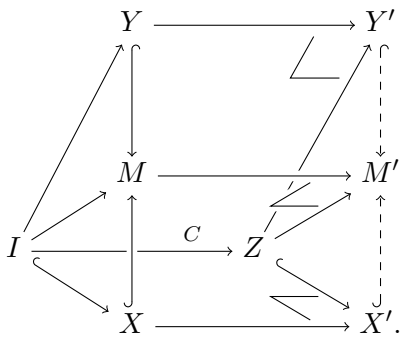
$$\begin{array}{ccc} n \cdot \star & \xrightarrow{[d_i]_{i \in n}} & [n+1] \\ \downarrow & & \downarrow \\ [n] & \longrightarrow & \nu_n. \end{array}$$

Definition 6 *A* move with interface *is one of these cospans with interface. The basic, full, or closed-world character is retained from the underlying move.*

Extended moves We now plug moves with interfaces into contexts, in the following sense.

Definition 7 *A* context *for a cospan with interface* (5) *is a position Z, equipped with a morphism $I \to Z$.*

From any cospan with interface μ as in (5) and context $C \colon I \to Z$, we construct the cospan $C[\mu]$ as in:



I.e., we push the available morphisms out of I along C, and infer the dashed morphisms, which form the desired cospan.

Definition 8 *An* extended move *is a cospan of the shape $C[\mu]$, for any move with interface μ and context C as above.*

Example 9 *Recall that* $[2]$ *is a position with one player knowing two channels. Recall from Figure 2 the pushout*

$$\begin{array}{ccc} \star & \xrightarrow{d_1} & [2] \\ {\scriptstyle d_2}\downarrow & & \downarrow{\scriptstyle p_2} \\ [2] & \xrightarrow[p_1]{} & 2\ {}_2|_1\ 2, \end{array}$$

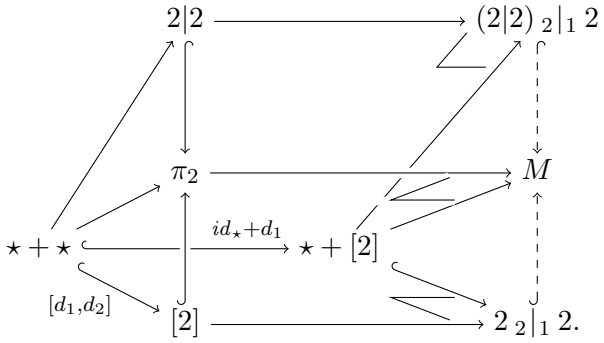
equivalently obtained as the pushout

$$\begin{array}{ccc} \star+\star & \xrightarrow{id_\star+d_1} & \star+[2] \\ {\scriptstyle [d_1,d_2]}\downarrow & & \downarrow{\scriptstyle [a_1,p_2]} \\ [2] & \xrightarrow[p_1]{} & 2\ {}_2|_1\ 2. \end{array}$$

The base position of Figure 1 is thus $2\ {}_2|_1\ 2$*. Recall also from* (4) *that* $2|2$ *denotes the position with two players both knowing two channels. Now, we have the forking move* $[2] \hookrightarrow \pi_2 \hookleftarrow 2|2$*. Equipping it with the interface*

$$[d_1, d_2] \colon \star+\star \to [2],$$

and putting it in the context $id_\star + d_1 \colon \star+\star \to \star+[2]$*, (which happens to be the same as the interface), we obtain*



This formally constructs the first layer of Figure 1. Constructing the whole play would be a little too verbose to be included here, but essentially straightforward.

Plays and relativisation We may now read Definition 1 again, this time in the formal setting, to define plays. Similarly, the definition of morphisms now makes rigorous sense, as well as Proposition 1.

Proof of Proposition 1: $\mathbb{E}$ is the full subcategory of the arrow category of $\widehat{\mathbb{C}}$ whose objects are plays. □

Similarly, Section 2.4 now makes rigorous sense.

4 Innocent Strategies as Sheaves

Now that the category of plays is defined, we move on to defining innocent strategies. There is a notion of a Grothendieck *site* [32], which consists of a category equipped with a (generalised) topology. On such sites, one may define a category of sheaves, which are very roughly the presheaves that are determined locally w.r.t. the generalised topology. We claim that there is a topology on each $\mathbb{E}_X$, for which sheaves adequately model innocent strategies. Fortunately, in our setting, sheaves admit a simple description, so that we can avoid the whole machinery. But sheaves were the way we arrived at the main ideas presented here, because they convey the right intuition: plays form a Grothendieck site, and the states of innocent strategies should be determined locally.

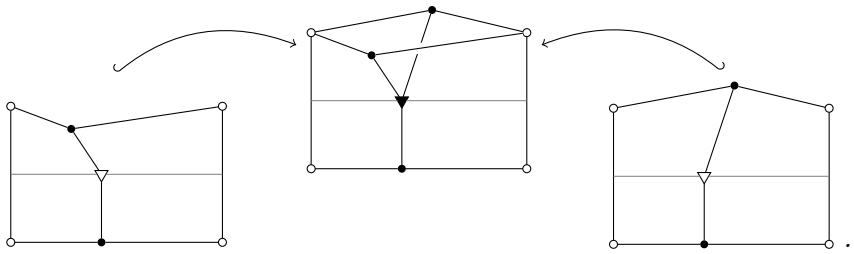
In this section, we first define innocent strategies, and state the spatial and temporal decomposition theorems. We then present our coalgebraic interpretation of innocent strategies, i.e., we define a polynomial endofunctor F, and show that presheaves of finite ordinals on views form a final F-coalgebra. We then derive from this a formal language and its interpretation in terms of innocent strategies. We finally use this language to translate CCS with recursive equations into innocent strategies.

4.1 Innocent Strategies

Definition 9 *A* view *is a finite, possibly empty 'composition'* $[n] \hookrightarrow V$ *of (extended) basic moves in* $\mathsf{Cospan}(\widehat{\mathbb{C}})$*, i.e., a play in which all the cospans are basic moves.*

When the composition is empty, we obtain $[n] \hookrightarrow [n]$, the *identity* view on $[n]$. We also note in passing that empty presheaves cannot be views, i.e., $X \hookrightarrow \emptyset$ is never a view.

Example 10 *Forking* (1) *has two non-trivial views, namely the (left legs of) basic moves* (2)*. Each of them embeds into forking:*



Example 11 *In Figure 1, the leftmost branch contains a view consisting of three basic moves: two π_2^l and an output.*

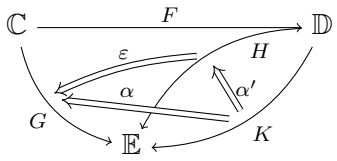
Definition 10 *For any position X, let $\mathbb{V}_X$ be the full subcategory of $\mathbb{E}_X$ consisting of views.*

More precisely, $\mathbb{V}_X$ consists of spans $U \hookleftarrow Y \to X$ where $Y \hookrightarrow U$ is a view.

Definition 11 *Let the category S_X of* innocent strategies *on X be the category $\widehat{\mathbb{V}_X}$ of presheaves on $\mathbb{V}_X$.*

A possible interpretation is that for a presheaf $F \in \widehat{\mathbb{V}_X}$ and view $V \in \mathbb{V}_X$, $F(V)$ is the set of possible *states* of the strategy F after playing V.

It might thus seem that we could content ourselves with defining only views, as opposed to plays. However, in order to define interactive equivalences in Section 5, we need to view innocent strategies as (particular) presheaves on the whole of $\mathbb{E}_X$.



The connection is as follows. Recall from MacLane [31] the notion of *right Kan extension*. Given functors F and G as on the right, a right Kan extension $\mathrm{Ran}_F(G)$ of G along F is a functor $H\colon \mathbb{D} \to \mathbb{E}$, equipped with a natural transformation $\varepsilon\colon HF \to G$, such that for all functors $K\colon \mathbb{D} \to \mathbb{E}$ and transformations $\alpha\colon KF \to G$, there is a unique $\alpha'\colon K \to H$ such that $\alpha = \varepsilon \bullet (\alpha' \circ id_F)$, where $\bullet$ is vertical composition of natural transformations. Now, precomposition with F induces a functor $\mathsf{Cat}(F, \mathbb{E})\colon \mathsf{Cat}(\mathbb{D}, \mathbb{E}) \to \mathsf{Cat}(\mathbb{C}, \mathbb{E})$, where $\mathsf{Cat}(\mathbb{D}, \mathbb{E})$ is the category of functors $\mathbb{D} \to \mathbb{E}$ and natural transformations between them. When $\mathbb{E}$ is complete, right Kan extensions always exist (and an explicit formula for our setting is given below), and choosing one of them for each functor $\mathbb{C} \to \mathbb{E}$ induces a right adjoint to $\mathsf{Cat}(F, \mathbb{E})$. Furthermore, it is known that when F is full and faithful, then ε is a natural isomorphism, i.e., $HF \cong G$.

Proposition 2 *If F is full and faithful, then* Ran_F *is a full essential embedding.*

Proof: First, let us show that Ran_F is essentially injective on objects. Indeed, assume $H = \mathrm{Ran}_F(G)$, $\mathrm{Ran}_F(G') = H'$, and $i\colon H \to H'$ is an isomorphism with inverse k. We must construct an isomorphism $G \cong G'$. Let $j\colon G \to G'$ be $\varepsilon_{G'} \bullet (iF) \bullet \varepsilon_G^{-1}$. Similarly, let $l\colon G' \to G$ be $\varepsilon_G \bullet (kF) \bullet \varepsilon_{G'}^{-1}$. We have

$$\begin{array}{rcl} l \bullet j & = & \varepsilon_G \bullet (kF) \bullet \varepsilon_{G'}^{-1} \bullet \varepsilon_{G'} \bullet (iF) \bullet \varepsilon_G^{-1} \\ & = & \varepsilon_G \bullet (kF) \bullet (iF) \bullet \varepsilon_G^{-1} \\ & = & \varepsilon_G \bullet ((k \bullet i) \circ F) \bullet \varepsilon_G^{-1} \\ & = & \varepsilon_G \bullet \varepsilon_G^{-1} \\ & = & id_G. \end{array}$$

Similarly, $j \bullet l = id_{G'}$ and we have $G \cong G'$.

To see that Ran_F is full, observe that for any $i\colon H \to H'$, with $H = \mathrm{Ran}_F(G)$ and $H' = \mathrm{Ran}_F(G')$, $j = \varepsilon_{G'} \bullet (iF) \bullet \varepsilon_G^{-1}$ is an antecedent of i by Ran_F. Indeed, by definition, $\mathrm{Ran}_F(j)$ is the unique $i'\colon H \to H'$ such that $\varepsilon_{G'} \bullet (i'F) = j \bullet \varepsilon_G$. But the latter is equal to $\varepsilon_{G'} \bullet (iF)$, so $i' = i$.

Finally, to show that Ran_F is faithful, consider $G, G'\colon \mathbb{C} \to \mathbb{E}$ and two natural transformations $i, j\colon G \to G'$ such that $\mathrm{Ran}_F(i) = \mathrm{Ran}_F(j) = k$. Then, by construction of k, we have

$$i \bullet \varepsilon_G = \varepsilon_{G'} \bullet (kF) = j \bullet \varepsilon_G.$$

But, ε_G being an isomorphism, this implies $i = j$ as desired. □

Returning to views and plays, the embedding $i_X\colon \mathbb{V}_X \hookrightarrow \mathbb{E}_X$ is full, so right Kan extension along $i_X^{op}\colon \mathbb{V}_X^{op} \to \mathbb{E}_X^{op}$ induces a full essential embedding $\mathrm{Ran}_{i_X^{op}}\colon \widehat{\mathbb{V}_X} \to \widehat{\mathbb{E}_X}$. The (co)restriction of this essential embedding to its essential image thus yields an essentially surjective, fully faithful functor, i.e., an equivalence of categories:

Proposition 3 *The category* S_X *is equivalent to the essential image of* $\mathrm{Ran}_{i_X^{op}}$.

The standard characterisation of right Kan extensions as ends [31] yields, for any $F \in \widehat{\mathbb{V}_X}$ and $U \in \mathbb{E}_X$:

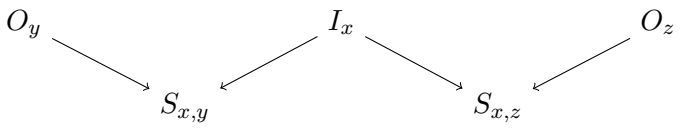
$$\mathrm{Ran}_{i_X^{op}}(F)(U) = \int_{V \in \mathbb{V}_X} F(V)^{\mathbb{E}_X(V,U)},$$

i.e., giving an element of $\mathrm{Ran}_{i_X^{op}}(F)$ on a play U amounts to giving, for each view V and morphism $V \to U$, an element of $F(V)$, satisfying some compatibility conditions. In Example 12 below, we compute an example right Kan extension.

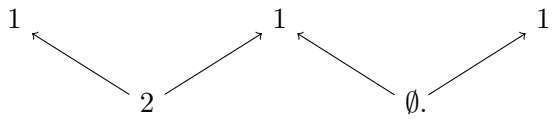
The interpretation of strategies in terms of states extends: for any presheaf $F \in \widehat{\mathbb{E}_X}$ and play $U \in \mathbb{E}_X$, $F(U)$ is the set of possible *states* of the strategy F after playing U. That F is in the image of $\mathrm{Ran}_{i_X^{op}}$ amounts to $F(U)$ being a compatible tuple of states of F after playing each view of U.

Example 12 *Here is an example of a presheaf $F \in \widehat{\mathbb{E}_X}$ which is not innocent, i.e., not in the image of $\mathrm{Ran}_{i_X^{op}}$. Consider the position X consisting of three players, say x, y, z, sharing a channel, say a. Let X_x be the subposition with only x and a, and similarly for X_y, X_z, $X_{x,y}$, and $X_{x,z}$. Let $I_x = (\iota_{1,1}^- \hookleftarrow X_x \hookrightarrow X)$ be the play where x inputs on a, and similarly let O_y and O_z be the plays where y and z output on a, respectively. Let now $S_{x,y} = (\tau_{1,1,1,1} \hookleftarrow X_{x,y} \hookrightarrow X)$ be the play where x and y synchronise on a (x inputs and y outputs), and similarly let $S_{x,z}$ be the play where x and z synchronise on a.*

Finally, we define a presheaf F on $\mathbb{E}/X$ such that $F(S_{x,y}) = 2$ is a two-element set, and $F(S_{x,z}) = \emptyset$. To define F on other plays, the idea is to map any strict subplay of $S_{x,y}$ and $S_{x,z}$ to a one-element set 1, and other plays to $\emptyset$. The cleanest technical way to do this seems to be as follows. The poset Ex_{11} defined by



fully embeds into $\mathbb{E}/X$, via, say i_{11}. Let F_0 be the presheaf on Ex_{11} defined by:



We now let $F = \mathrm{Ran}_{i_{11}^{op}}(F_0)$. Because i_{11} is fully faithful, F coincides with F_0 on the plays of Ex_{11}, as desired.

Now, F fails to be innocent on two counts. First, since x and y accept to input and output in only one way, it is non-innocent to accept that they

synchronise in more than one way. Formally, $S_{x,y}$ has two non-trivial views, I_x and O_y, so since F maps identity views to a singleton, $F(S_{x,y})$ should be isomorphic to $F(I_x) \times F(O_y) = 1 \times 1 = 1$. The second reason why F is not innocent is that, since x and z accept to input and output, F should accept that they synchronise. Formally, $F(S_{x,z})$ should also be a singleton. This altogether models the fact that in CCS, processes do not get to know with which other processes they synchronise.

The restriction of F to $\mathbb{V}_X$, i.e., $F' = F \circ i_X^{op}$, in turn has a right Kan extension F'', which is innocent. (In passing, the unit of the adjunction $\mathsf{Cat}(i_X^{op}, \mathsf{Set}) \dashv \mathrm{Ran}_{i_X^{op}}$ is a natural transformation $F \to F''$.) To conclude this example, let us compute F''. First, F' only retains from F its values on views. So, if X_x denotes the empty view on X_x, $F'(X_x) = 1$, and similarly $F'(X_y) = F'(X_z) = 1$. Furthermore, $F'(I_x) = F'(O_y) = F'(O_z) = 1$. Finally, for any view V not isomorphic to any of the previous ones, $F'(V) = \emptyset$. So, recall that F'' maps any play $U \hookleftarrow Y \hookrightarrow X$ to $\int_{V \in \mathbb{V}_X} F'(V)^{\mathbb{E}_X(V,U)}$. So, e.g., since the views of $S_{x,y}$ are subviews of I_x and O_y, we have $F''(S_{x,y}) = F'(I_x) \times F'(O_y) = 1$. Similarly, $F''(S_{x,z}) = 1$. But also, for any play U such that all views $V \to U$ are subviews of either of I_x, O_y, or O_z, we have $F''(U) = 1$. Finally, for any play U such that there exists a view $V \to U$ which is not a subview of any of I_x, O_y, or O_z, we have $F''(U) = \emptyset$.

One way to understand Proposition 3 is to view $\widehat{\mathbb{V}_X}$ as the syntax for innocent strategies: presheaves on views are (almost) infinite terms in a certain syntax (see Section 4.4 below). On the other hand, seeing them as presheaves on plays will allow us to consider their global behaviour: see Section 5 when we restrict to the closed-world game. Thus, right Kan extension followed by restriction to closed-world will associate a semantics to innocent strategies.

Remark 5 *The relevant Grothendieck topology on $\mathbb{E}_X$ says, roughly, that a play is covered by its views. Any sheaf for this topology is determined by its restriction to $\mathbb{V}_X$, for its elements on any non-view play U are precisely amalgamations of its elements on views of U. Right Kan extension just computes these amalgamations in the particular case of a topology derived from a full subcategory, here views.*

So, we have defined for each X the category S_X of innocent strategies on X. This assignment is actually functorial $\mathbb{B}^{op} \to \mathsf{CAT}$, as follows (where CAT is the large category of locally small categories). Any morphism $f : Y \to X$ induces a functor $f_! : \mathbb{V}_Y \to \mathbb{V}_X$ mapping $(V \hookleftarrow Z \to Y)$ to

$(V \leftarrow Z \rightarrow Y \rightarrow X)$. Precomposition with $(f_!)^{op}$ thus induces a functor $\mathsf{S}_f \colon \widehat{\mathbb{V}_X} \to \widehat{\mathbb{V}_Y}$.

Proposition 4 *This defines a functor* $\mathsf{S} \colon \mathbb{B}^{op} \to \mathsf{CAT}$.

Proof: A straightforward verification. □

But there is more: for any position, giving a strategy for each player in it easily yields a strategy on the whole position. We call this *amalgamation* of innocent strategies (because the functor S is indeed a *stack* [43], and this is a particular case of amalgamation in that stack). Formally, consider any subpositions X_1 and X_2 of a given position X, inducing a partition of the players of X, i.e., such that $X_1 \cup X_2$ contains all players of X, and $X_1 \cap X_2$ contains none. Then $\mathbb{V}_X$ is isomorphic to the coproduct $\mathbb{V}_{X_1} + \mathbb{V}_{X_2}$. (Indeed, a view contains in particular an initial player in X, which forces it to belong either in $\mathbb{V}_{X_1}$ or in $\mathbb{V}_{X_2}$.)

Definition 12 *Given innocent strategies* F_1 *on* X_1 *and* F_2 *on* X_2*, let their* amalgamation *be their copairing*

$$[F_1, F_2] : \mathbb{V}_X^{op} \cong (\mathbb{V}_{X_1} + \mathbb{V}_{X_2})^{op} \cong \mathbb{V}_{X_1}^{op} + \mathbb{V}_{X_2}^{op} \to \mathsf{Set}.$$

By universal property of coproduct:

Proposition 5 *Amalgamation yields an isomorphism of categories*

$$\widehat{\mathbb{V}_X} \cong \widehat{\mathbb{V}_{X_1}} \times \widehat{\mathbb{V}_{X_2}}.$$

Example 13 *Consider again the position* X *from Example 12, and let* $X_{y,z}$ *be the subposition with only* y *and* z*. We have* $\mathbb{V}_X \simeq (\mathbb{V}_{X_x} + \mathbb{V}_{X_{y,z}})$*, which we may explain by hand as follows. A view on* X *has a base player,* x*,* y*, or* z*, and so belongs either in* $\mathbb{V}_{X_x}$ *or in* $\mathbb{V}_{X_{y,z}}$*. Furthermore, if* V *is a view on* x *and* W *is a view on* y*, then* $\mathbb{V}_X(V, W) = \emptyset$ *(and similarly for any pair of distinct players in* X*).*

Now, recall F'*, the restriction of* F *to* $\mathbb{V}_X$*. We may define* $F_x \colon \mathbb{V}_{X_x}^{op} \to \mathsf{Set}$ *to be the restriction of* F' *along the (opposite of the) embedding* $\mathbb{V}_{X_x} \hookrightarrow \mathbb{V}_X$*, and similarly* $F_{y,z}$ *to be the restriction of* F' *along* $\mathbb{V}_{X_{y,z}} \hookrightarrow \mathbb{V}_X$*. We have obviously* $F' = [F_x, F_{y,z}]$.

Analogous reasoning leads to what we call spatial decomposition. For any X, let $\mathrm{Pl}(X) = \sum_n X([n])$, i.e., the set of pairs (n, x), where x is a player in X, knowing n channels.

Theorem 1 *We have* $\widehat{\mathbb{V}_X} \cong \prod_{(n,x) \in \mathrm{Pl}(X)} \widehat{\mathbb{V}_{[n]}}$.

Again, this is a particular case of amalgamation in the stack S, but we do not need to spell out the definition here.

4.2 Temporal Decomposition

Let us now describe *temporal* decomposition. Recall that basic moves are left and right half-forking (2), input, output, tick, and channel creation.

Definition 13 *Let $\mathcal{M}$ be the graph with vertices all natural numbers n, and with edges $n \to n'$ all (isomorphism classes of) basic moves $M\colon [n] \to [n']$.*

Recall from Remark 4 that the notion of isomorphism considered here is that of an isomorphism of cospans in $\widehat{\mathbb{C}}$.

Definition 14 *Let $\mathcal{M}_n$ be the set of edges from n in $\mathcal{M}$.*

For stating the temporal decomposition theorem, we need a standard [21] categorical construction, the category of families on a given category $\mathbb{C}$. First, given a set X, consider the category $\mathrm{Fam}(X)$ with as objects X-indexed families of sets $Y = (Y_x)_{x\in X}$, and as morphisms $Y \to Z$ families $(f_x\colon Y_x \to Z_x)_{x\in X}$ of maps. This category is equivalently described as the slice category Set/X. To see the correspondence, consider any family $(Y_x)_{x\in X}$, and map it to the projection function $\sum_{x\in X} Y_x \to X$ sending (x, y) to x. Conversely, given $f\colon Y \to X$, let, for any $x \in X$, Y_x be the fibre of f over x, i.e., $f^{-1}(x)$.

Generalising from sets X to small categories $\mathbb{C}$, $\mathrm{Fam}(\mathbb{C})$ has as objects families $p\colon Y \to \mathrm{ob}(\mathbb{C})$ indexed by the objects of $\mathbb{C}$. Morphisms $(Y, p) \to (Z, q)$ are pairs of $u\colon Y \to Z$ and $v\colon Y \to \mathrm{mor}(\mathbb{C})$, where $\mathrm{mor}(\mathbb{C})$ is the set of morphisms of $\mathbb{C}$, such that $\mathrm{dom} \circ v = p$, and $\mathrm{cod} \circ v = q \circ u$. Thus, any element $y \in Y$ over $C \in \mathbb{C}$ is mapped to some $u(y) \in Z$ over $C' \in \mathbb{C}$, and this mapping is labelled by a morphism $v(y)\colon C \to C'$ in $\mathbb{C}$. The obtained category is locally small.

Further generalising, for $\mathbb{C}$ a locally small category, we may define $\mathrm{Fam}(\mathbb{C})$ in exactly the same way (with Y still a set), and the obtained category remains locally small.

The temporal decomposition theorem is:

Theorem 2 *There is an equivalence of categories*

$$\mathsf{S}_n \simeq \mathrm{Fam}\left(\prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right).$$

The main intuition is that an innocent strategy is determined up to isomorphism by (i) its initial states, and (ii) what remains of them after each possible basic move. The family construction is what permits innocent strategies with several possible states over the identity play.

Proof sketch: For general reasons, we have:

$$\begin{array}{rcl} \mathrm{Fam}\left(\prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right) & = & \mathrm{Fam}\left(\prod_{M\in\mathcal{M}_n} [\mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]\right) \\ & \cong & \mathrm{Fam}\left(\left[\sum_{M\in\mathcal{M}_n} \mathbb{V}_{\mathrm{cod}(M)}{}^{op}, \mathsf{Set}\right]\right) \\ & \simeq & \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \downarrow \Delta, \end{array}$$

where $\Delta\colon \mathsf{Set} \to [\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$ maps any set X to the constant presheaf mapping any object to X and any morphism to the identity.

By definition, the last category is a lax pullback

$$\begin{array}{ccc} \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] & = & \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \\ {\scriptstyle\Delta}\uparrow & \swarrow & \uparrow \\ \mathsf{Set} & \longleftarrow & \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \downarrow \Delta \end{array}$$

in CAT.

Now, any basic move $M\colon n \to n'$ induces a functor $(-\circ M)\colon \mathbb{V}_{[n']} \to \mathbb{V}_{[n]}$, mapping any view $V \in \mathbb{V}_{[n']}$ to $V \circ M$ (with composition in $\mathsf{Cospan}(\widehat{\mathbb{C}})$). We show that the square

$$\begin{array}{ccc} \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} & = & \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} \\ {\scriptstyle !}\downarrow & \swarrow^{\lambda} & \downarrow{\scriptstyle [-\circ M]_{M\in\mathcal{M}_n}} \\ 1 & \xrightarrow[\ulcorner id_{[n]} \urcorner]{} & \mathbb{V}^{op}_{[n]} \end{array} \qquad (6)$$

is a lax pushout in Cat, where $\lambda_{M,V}\colon id_{[n]} \to M \circ V$, seen in $\mathbb{V}_{[n]}$, is the obvious inclusion, which for general reasons is mapped by the hom-2-functor $\mathsf{CAT}(-, \mathsf{Set})$ to a lax pullback. But $\mathsf{CAT}(!, \mathsf{Set}) = \Delta$ and $\mathsf{CAT}(id, \mathsf{Set}) = id$, so we obtain a canonical isomorphism of lax pullbacks

$$\mathsf{S}_{[n]} = [\mathbb{V}^{op}_{[n]}, \mathsf{Set}] \cong \left[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}\right] \downarrow \Delta.$$

More detail is in Appendix A. □

Remark 6 *The theorem almost makes innocent strategies into a* sketch *(on the category with positions as objects, finite compositions of extended moves as morphisms, and the $\mathcal{M}_X$'s as distinguished cones). Briefly, being a sketch would require a bijection of sets $\mathsf{S}_n \cong \prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}$. Here, the bijection becomes an equivalence of categories, and the family construction sneaks in.*

4.3 Innocent Strategies as a Terminal Coalgebra

Temporal decomposition gives

$$\mathsf{S}_n \simeq \mathrm{Fam}\left(\prod_{M\in\mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right),$$

for all n. Considering a variant of this formula as a system of equations will lead to our interpretation of CCS. The first step is to replace Set with FinOrd, the category of finite ordinals and monotone functions. The proof applies *mutatis mutandis* and we obtain an equivalence, which, because both categories are skeletal, is an isomorphism:

$$\widehat{\mathbb{V}_{[n]}} \cong \mathrm{Fam}_f\left(\prod_{M\in\mathcal{M}_n} \widehat{\mathbb{V}_{\mathrm{cod}(M)}}\right), \tag{7}$$

where

- Fam_f is the same as Fam but with finite families, i.e., for any category $\mathbb{C}$, $\mathrm{ob}(\mathrm{Fam}_f(\mathbb{C})) = \sum_{I\in\mathsf{FinOrd}}(\mathrm{ob}(\mathbb{C}))^I = (\mathrm{ob}(\mathbb{C}))^*$ is the set of finite words over objects of $\mathbb{C}$, also known as the free monoid on $\mathrm{ob}(\mathbb{C})$;
- and for any category $\mathbb{C}$, $\widehat{\mathbb{C}}$ denotes the functor category $[\mathbb{C}^{op}, \mathsf{FinOrd}]$.

Remark 7 *Recall that in the proof of Theorem 2,* Fam *arises from the 'constant presheaf' functor $\Delta\colon \mathsf{Set} \to \widehat{-}$, with $-$ a complicated category. This functor itself is equal to restriction along $- \to 1$, via $\widehat{1} \cong \mathsf{Set}$. Replacing* Set *with* FinOrd *thus replaces Δ with the analogous functor $\mathsf{FinOrd} \to \widehat{-}$, via $\widehat{1} \cong \mathsf{FinOrd}$, and thus* Fam *with* Fam_f.

Furthermore, because FinOrd *embeds into* Set*, the special strategies of $\widehat{\mathbb{V}_{[n]}}$ embed into $\mathsf{S}_{[n]}$.*

Then, taking advantage of the fact that FinOrd is a small category, we consider its set FinOrd_0 of objects, i.e., finite ordinals, and the endofunctor

F on $\mathsf{Set}/\mathsf{FinOrd}_0$ defined on any family of sets $X = (X_i)_{i \in \mathsf{FinOrd}_0}$ by:

$$(\mathsf{F}(X))_n = \sum_{I \in \mathsf{FinOrd}_0} \left(\prod_{M \in \mathcal{M}_n} X_{\mathrm{cod}(M)} \right)^I,$$

where we abusively confuse $[n'] = \mathrm{cod}(M)$ and the natural number n' itself. The isomorphism (7) becomes

$$\mathrm{ob}(\widehat{\mathbb{V}_{[n]}}) \cong (\mathsf{F}(\mathrm{ob}(\widehat{\mathbb{V}_{-}})))_n.$$

We may decompose F as follows. Consider the endofunctor on $\mathsf{Set}/\mathsf{FinOrd}_0$ defined by $(\partial X)_n = \prod_{M \in \mathcal{M}_n} X_{\mathrm{cod}(M)}$, for any family X. We obviously have:

Lemma 1 *F is equal to the composite $(\partial -)^*$.*

This endofunctor is polynomial [27] and we now give a characterisation of its final coalgebra. The rest of this subsection is devoted to proving:

Theorem 3 *The family $\mathrm{ob}(\widehat{\mathbb{V}_n})$ formed for each n by (the objects of) $\widehat{\mathbb{V}_n}$ is a terminal coalgebra for F.*

Consider any F-coalgebra $a \colon X \to \mathsf{F}X$.

We define by induction on N a sequence of maps $f_N \colon X \to \widehat{\mathbb{V}_{[-]}}$, such that for any view V of length less than N (i.e., with less than N basic moves), and any $N' > N$, $f_{N'}(x)(V) = f_N(x)(V)$, and similarly the action of $f_N(x)$ on morphisms is the same as that of $f_{N'}(x)$.

To start the induction, take $f_0(x)$ to be the strategy mapping $id_{[n]}$ to $\pi(a(x))$, i.e., the length of $a(x) \in \sum_{I \in \mathsf{FinOrd}_0} ((\partial X)_n)^I$, and all other views to 0.

Furthermore, given f_N, define f_{N+1} to be

$$X \xrightarrow{a} \mathsf{F}X \xrightarrow{\mathsf{F}(f_N)} \mathsf{F}(\widehat{\mathbb{V}_{[-]}}) \xrightarrow{\cong} \widehat{\mathbb{V}_{[-]}},$$

where the equivalence is by temporal decomposition.

Unfolding the definitions yields:

Lemma 2 *Consider any $x \in X_n$, and $a(x) = (z_1, \ldots, z_k)$. For any move $M \colon n \to n'$ and view $V \colon n' \to n''$ of length at most N, and for any $i \in k$, $f_{N+1}(x)(V \circ M) = \sum_{i \in k} f_N(z_i(M))(V)$.*

For any $x \in X_n$, we have a sequence $f_0(x) \hookrightarrow f_1(x) \hookrightarrow \dots f_N(x) \hookrightarrow f_{N+1}(x) \hookrightarrow \dots$ which is pointwise stationary. This sequence thus has a colimit in $\widehat{\mathbb{V}_{[n]}}$, the presheaf mapping any view V of length N to $f_N(V)$ (or equivalently $f_{N'}(V)$ for any $N' \geq N$), which allows us to define:

Definition 15 *Let $f\colon X \to \widehat{\mathbb{V}_{[-]}}$ map any $x \in X_n$ to $\bigcup_N f_N(x)$.*

By construction, we have

Lemma 3 *The following diagram commutes:*

$$\begin{array}{ccc} X & \xrightarrow{\quad a \quad} & \mathsf{F}X \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle \mathsf{F}(f)} \\ \widehat{\mathbb{V}_{[-]}} & \xleftarrow{\quad\cong\quad} & \mathsf{F}(\widehat{\mathbb{V}_{[-]}}). \end{array}$$

Lemma 4 *The map f is a morphism of F-coalgebras.*

Proof: Let, for any innocent strategy $S \in \widehat{\mathbb{V}_{[n]}}$ and $i \in S(id_{[n]})$, $S_{|i}$ be the strategy mapping any view V to the fibre over i of $S(V) \to S(id_{[n]})$. Using the notations of Lemma 2, we must show that for any $i \in k$, we have $(f(x))_{|i}(V \circ M) = f(z_i(M))(V)$. But Lemma 2 entails that $f(x)(V \circ M) \to f(x)(id_{[n]})$ is actually the coproduct over $i' \in k$ of all $f(z_{i'}(M))(V) \xrightarrow{!} 1 \xrightarrow{i'} \pi(a(x))$, so its fibre over i is indeed $f(z_i(M))(V)$. □

Lemma 5 *The map f is the unique map $X \to \widehat{\mathbb{V}_{[-]}}$ of F-coalgebras.*

Proof: Consider any such map g of coalgebras. It must be such that $g(x)(id_{[n]}) = \pi(a(x))$, and furthermore, using the same notation as before, for any $i \in k$ $(g(x))_{|i}(V \circ M) = g(z_i(M))(V)$, which imposes by induction that $f = g$. □

The last two lemmas directly entail Theorem 3.

4.4 Languages

A consequence of Theorem 3 is that the family $\widehat{\mathbb{V}_n}$ supports the operations of the grammar

$$\frac{\dots\ n \vdash F_i\ \dots\ (\forall i \in I)}{n \vdash \sum_{i \in I} F_i}\ (I \in \mathsf{FinOrd}_0)$$

$$\frac{\dots\ n' \vdash F_M\ \dots\ (\forall M\colon [n] \to [n'] \in \mathcal{M})}{n \vdash \langle M \mapsto F_M \rangle}\ .$$

Here, $n \vdash F$ denotes a presheaf of finite ordinals on $\mathbb{V}_n$. The interpretation is as follows: given presheaves $F_1, \ldots, F_I$, for $I \in \mathsf{FinOrd}_0$, the first rule constructs the finite coproduct $\sum_{i \in I} F_i$ of presheaves (finite coproducts exist in $\widehat{\mathbb{V}_n}$ because they do in FinOrd). In particular, when I is the empty ordinal, we sum over an empty set, so the rule degenerates to

$$\frac{}{n \vdash \emptyset} \ .$$

In terms of presheaves, this is just the constantly empty presheaf.

For the second rule, if for all basic $M \colon [n] \to [n']$, we are given $F_M \in \widehat{\mathbb{V}_{[n']}}$, then $\langle M \mapsto F_M \rangle$ denotes the image under (7) of

$$(1, 1 \mapsto M \mapsto F_M).$$

Here, we provide an element of the right-hand side of (7), consisting of the finite ordinal $I = 1 = \{1\}$, and the function mapping M to $F_M \in \widehat{\mathbb{V}_{[n']}}$ (up to currying). That was for parsing; the intuition is that we construct a presheaf with one initial state, 1, which maps any view starting with M, say $V \circ M$, to $F_M(V)$. Thus the F_M's specify what remains of our presheaf after each possible basic move. In particular, when all the F_M's are empty, we obtain a presheaf which has an initial state, but which does nothing beyond it. We abbreviate it as $0 = \langle \_ \mapsto \emptyset \rangle$.

4.5 Translating CCS

It is rather easy to translate CCS into this language. First, define CCS syntax by the natural deduction rules in Figure 3, where Names and Vars are two fixed, disjoint, and infinite sets of *names* and *variables*; Ξ ranges over finite sequences of pairs $(x \colon n)$ of a variable x and its *arity* $n \in \mathsf{FinOrd}_0$, such that the variables are pairwise distinct; Γ ranges over finite sequences of pairwise distinct names; there are two judgements: $\Gamma \vdash P$ for *global* processes, $\Xi; \Gamma \vdash P$ for *open* processes. Rule Global is the only rule for forming global processes, and there $\Xi = (x_1 \colon |\Delta_1|, \ldots, x_n \colon |\Delta_n|)$. Finally, α denotes a or $\overline{a}$, for $a \in \mathsf{Names}$, and $\lfloor a \rfloor = \lfloor \overline{a} \rfloor = a$.

First, we define the following (approximation of a) translation on open processes, mapping each open process $\Xi; \Gamma \vdash P$ to $[\![P]\!] \in \widehat{\mathbb{V}_n}$, for $n = |\Gamma|$. This translation ignores the recursive definitions, and we will refine it below to take them into account. We proceed by induction on P, leaving contexts

$$\textsc{CCSApp}\quad \frac{}{\Xi;\Gamma \vdash x(a_1,\dots,a_n)}\ ((x\colon n)\in\Xi \text{ and } a_1,\dots,a_n\in\Gamma)$$

$$\frac{\Xi;\Gamma,a\vdash P}{\Xi;\Gamma\vdash \nu a.P}\ (a\notin\Gamma) \qquad \frac{\Xi;\Gamma\vdash P \qquad \Xi;\Gamma\vdash Q}{\Xi;\Gamma\vdash P|Q}$$

$$\frac{\dots \qquad \Xi;\Gamma\vdash P_i \qquad \dots \qquad (\forall i\in I)}{\Xi;\Gamma\vdash \sum_{i\in I}\alpha_i.P_i}\ (I\in\mathsf{FinOrd}_0 \text{ and } \forall i\in I, \lfloor\alpha_i\rfloor\in\Gamma)$$

$$\textsc{Global}\quad \frac{\Xi;\Delta_1\vdash P_1 \qquad \dots \qquad \Xi;\Delta_n\vdash P_n \qquad \Xi;\Gamma\vdash P}{\Gamma\vdash \texttt{rec}\ x_1(\Delta_1)\ \texttt{:=}\ P_1,\dots,x_n(\Delta_n)\ \texttt{:=}\ P_n\ \texttt{in}\ P}$$

Figure 3: CCS syntax

$\Xi;\Gamma$ implicit:

$$x(a_1,\dots,a_k)\mapsto\emptyset \qquad \nu a.P\mapsto\langle \nu_n\mapsto[\![P]\!], \_\mapsto\emptyset\rangle$$

$$P|Q\mapsto\langle\ \pi^l_n\mapsto[\![P]\!],\ \pi^r_n\mapsto[\![Q]\!],\ \_\mapsto\emptyset\ \rangle \qquad \textstyle\sum_{i\in I}\alpha_i.P_i\mapsto\langle\ (\iota^+_{n,j}\mapsto\sum_{k\in I_j}[\![P_k]\!],\ \iota^-_{n,j}\mapsto\sum_{k\in I_j}[\![P_k]\!]\ )_{j\in n},\ \_\mapsto\emptyset\ \rangle.$$

Let us explain intuitions and notation. In the first case, we assume implicitly that $(x\colon k)\in\Xi$; the intuition is just that we approximate variables with empty strategies. Next, $P|Q$ is translated to the strategy with one initial state, which only accepts left and right half-forking first, and then lets its avatars play $[\![P]\!]$ and $[\![Q]\!]$, respectively. Similarly, $\nu a.P$ is translated to the strategy with one initial state, accepting only the channel creation move, and then playing $[\![P]\!]$. In the last case, the guarded sum $\sum_{i\in I}\alpha_i.P_i$ is translated to the strategy with one initial state, which

- accepts input on any channel a when $\alpha_i=a$ for some $i\in I$, and output on any channel a when $\alpha_i=\overline{a}$ for some $i\in I$;
- after an input on a, plays the sum of all $[\![P_i]\!]$'s such that $\alpha_i=a$; and after an output on a, plays the sum of all $[\![P_i]\!]$'s such that $\alpha_i=\overline{a}$.

Formally, in the definition, we let, for all $j \in n$, $I_{\overline{j}} = \{i \in I \mid \alpha_i = \overline{a_j}\}$ and $I_j = \{i \in I \mid \alpha_i = a_j\}$. In particular, if $I = \emptyset$, we obtain 0.

Thus, almost all translations of open processes have exactly one initial state, i.e., map the identity view on $[n]$ to the singleton 1. The only exceptions are variable applications, which are mapped to the empty presheaf.

The translation extends to global processes as follows. Fixing a global process $Q = (\texttt{rec}\ x_1(\Delta_1)\ \texttt{:=}\ P_1, \ldots, x_k(\Delta_k)\ \texttt{:=}\ P_k\ \texttt{in}\ P)$ typed in Γ with n names, define the sequence $(P^i)_{i \in \mathsf{FinOrd}_0}$ of open processes (all typed in $\Xi; \Gamma$) as follows. First, $P^0 = P$. Then, let $P^{i+1} = \mathsf{d}P^i$, where d is the *derivation* endomap on open processes typed in any extension $\Xi; (\Gamma, \Delta)$ of $\Xi; \Gamma$, which unfolds one layer of recursive definitions. This map is defined by induction on its argument as follows:

$$\mathsf{d}(x_l(a_1, \ldots, a_{k_l})) = P_l[b_j \mapsto a_j]_{1 \leq j \leq k_l} \qquad \mathsf{d}(\nu a.P) = \nu a.\mathsf{d}P$$
$$\mathsf{d}(P|Q) = \mathsf{d}P|\mathsf{d}Q \qquad \mathsf{d}(\textstyle\sum_{i \in I} \alpha_i.P_i) = \sum_{i \in I} \alpha_i.(\mathsf{d}P_i),$$

where for all $l \in \{1, \ldots, k\}$, $\Delta_l = (b_1, \ldots, b_{k_l})$, and $P[\sigma]$ denotes simultaneous, capture-avoiding substitution of names in P by σ.

By construction, the translations of these open processes form a sequence $[\![P^0]\!] \hookrightarrow [\![P^1]\!] \ldots$ of inclusions in $\widehat{\mathbb{V}_n}$, such that for any natural number i and view $V \in \mathbb{V}_n$ of length i, $[\![P^j]\!](V)$ is fixed after $j = (k+1)i$, at worst, i.e., for all $j \geq (k+1)i$, $[\![P^j]\!](V) = [\![P^{(k+1)i}]\!](V)$. Thus, this sequence has a colimit in $\widehat{\mathbb{V}_n}$, the presheaf sending any view V of length i to $[\![P^{(k+1)i}]\!](V)$. We put:

Definition 16 *Let the translation of Q be $[\![Q]\!] = \mathrm{colim}_{i \in \mathsf{FinOrd}} [\![P^i]\!]$.*

Which equivalence is induced by this mapping on CCS, especially when taking into account the interactive equivalences developed in the next section? This is the main question we will try to address in future work.

5 Interactive Equivalences

5.1 Fair Testing vs. Must Testing: The Standard Case

An important part of concurrency theory consists in studying *behavioural equivalences*. Since each such equivalence is supposed to define when two processes behave the same, it might seem paradoxical to consider several of them. Van Glabbeek [42] argues that each behavioural equivalence corresponds to a physical scenario for observing processes.

A distinction we wish to make here is between *fair* scenarios, and *potentially unfair* ones. An example of a fair scenario is when parallel composition of processes is thought of as modelling different physical agents, e.g., in a game with several players. Otherwise said, players are really independent. On the other hand, an example of a potentially unfair scenario is when parallelism is implemented via a scheduler.

This has consequences on so-called *testing* equivalences [7]. Let $\heartsuit$ be a fixed action.

Definition 17 *A process P is* must orthogonal *to a context C, notation $P \perp^m C$, when all maximal traces of $C[P]$ play $\heartsuit$ at some point.*

Here, maximal means either infinite or finite without extensions. Let $P^{\perp^m}$ be the set of all contexts must orthogonal to P.

Definition 18 *P and Q are* must equivalent*, notation $P \sim_m Q$, when $P^{\perp^m} = Q^{\perp^m}$.*

In transition systems, or automata, we have $\Omega \sim_m \Omega|\overline{a}$ (where Ω is the looping process, producing infinitely many silent transitions). This might be surprising, because the context $C = a.\heartsuit \mid \Box$ intuitively should distinguish these processes, by being orthogonal to $\Omega|\overline{a}$ but not to Ω alone. However, it is not orthogonal to $\Omega|\overline{a}$, because $C[\Omega|\overline{a}]$ has an infinite looping trace giving priority to Ω. This looping trace is unfair, because the synchronisation on a is never performed. Thus, one may view the equivalence $\Omega \sim_m \Omega|\overline{a}$ as exploiting potential unfairness of a hypothetical scheduler.

Usually, concurrency theorists consider this too coarse, and resort to *fair* testing equivalence.

Definition 19 *A process P is* fair orthogonal *to a context C, notation $P \perp^f C$, when all finite traces of $C[P]$ extend to traces that play $\heartsuit$ at some point.*

Again, $P^{\perp^f}$ denotes the set of all contexts fair orthogonal to P.

Definition 20 *P and Q are* fair equivalent*, notation $P \sim_f Q$, when $P^{\perp^f} = Q^{\perp^f}$.*

This solves the issue, i.e., $\Omega \nsim_f \Omega|\overline{a}$.

In summary, the mainstream setting for testing equivalences relies on traces; and the notion of maximality for traces is intrinsically unfair. This is

usually rectified by resorting to fair testing equivalence over must testing equivalence. Our setting is more flexible, in the sense that maximal plays are better behaved than maximal traces. In terms of the previous section, this allows viewing the looping trace $\Omega|\overline{a}|a.\heartsuit \xrightarrow{\tau} \Omega|\overline{a}|a.\heartsuit \xrightarrow{\tau} \dots$ as non-maximal. In the next sections, we define an abstract notion of interactive equivalence (still in the particular case of CCS but in our setting) and we instantiate it to define and study the counterparts of must and fair testing equivalences.

5.2 Interactive Equivalences

Definition 21 *A play is* closed-world *when it is a composite of closed-world extended moves.*

Equivalently, a play is closed-world when all of its basic moves are part of a closed-world move.

Let $\mathbb{W} \hookrightarrow \mathbb{E}$ be the full subcategory of closed-world plays, $\mathbb{W}(X)$ being the *fibre* over X for the projection functor $\mathbb{W} \to \mathbb{B}$, i.e., the subcategory of $\mathbb{W}$ consisting of closed-world plays with base X, and morphisms (id_X, k) between them[5].

Let the category of *closed-world behaviours* on X be the category $\mathsf{G}_X = \widehat{\mathbb{W}(X)}$ of presheaves on $\mathbb{W}(X)$. We may now put:

Definition 22 *An* observable criterion *consists for all positions X, of a replete subcategory $\perp\!\!\!\perp_X \hookrightarrow \mathsf{G}_X$.*

Recall that $\perp\!\!\!\perp_X$ being replete means that for all $F \in \perp\!\!\!\perp_X$ and isomorphism $f\colon F \to F'$ in G_X, F' and f are in $\perp\!\!\!\perp_X$.

An observable criterion specifies the class of 'successful', closed-world behaviours. The two criteria considered below are two ways of formalising the idea that a successful behaviour is one in which all accepted closed-world plays are 'successful', in the sense that some player plays the tick move at some point.

We now define interactive equivalences. Recall that $[F, G]$ denotes the amalgamation of F and G, and that right Kan extension along i_Z^{op} induces a functor $\mathrm{Ran}_{i_Z^{op}}\colon \widehat{\mathbb{V}_Z} \to \widehat{\mathbb{E}_Z}$. Furthermore, precomposition with the canonical

[5]This is not exactly equivalent to what could be noted $\mathbb{W}_X$, since in the latter there are objects $U \hookleftarrow Y \hookrightarrow X$ with a strict inclusion $Y \hookrightarrow X$. However, both should be equivalent for what we do in this paper, i.e., fair and must equivalences.

inclusion $j_Z \colon \mathbb{W}(Z) \hookrightarrow \mathbb{E}_Z$ induces a functor $j_Z^* \colon \widehat{\mathbb{E}_Z} \to \widehat{\mathbb{W}(Z)}$. Composing the two, we obtain a functor $\mathsf{Gl} \colon \mathsf{S}_Z \to \mathsf{G}_Z$:

$$\mathsf{S}_Z = \widehat{\mathbb{V}_Z} \xrightarrow{\mathrm{Ran}_{i_Z^{op}}} \widehat{\mathbb{E}_Z} \xrightarrow{j_Z^*} \widehat{\mathbb{W}(Z)} = \mathsf{G}_Z.$$

Definition 23 *For any innocent strategy F on X and any pushout square P of positions as on the right, with I consisting only of channels, let $F^{\perp\!\!\!\perp_P}$ be the class of all innocent strategies G on Y such that $\mathsf{Gl}([F,G]) \in \perp\!\!\!\perp_Z$.*

$$\begin{array}{ccc} I & \longrightarrow & Y \\ \downarrow & \ulcorner & \downarrow \\ X & \longrightarrow & Z \end{array} \qquad (8)$$

Here, G is thought of as a *test* for F. Also, P denotes the whole pushout square and $F^{\perp\!\!\!\perp_P}$ denotes all the valid tests for the considered pushout square P. From the CCS point of view, I corresponds to the set of names shared by the process under observation (F) and the test (G).

Definition 24 *Any two innocent strategies $F, F' \in \mathsf{S}_X$ are* $\perp\!\!\!\perp$-equivalent*, notation $F \sim_{\perp\!\!\!\perp} F'$, iff for all pushouts P as in 8, $F^{\perp\!\!\!\perp_P} = F'^{\perp\!\!\!\perp_P}$.*

5.3 Fair vs. Must

Let us now define fair and must testing equivalences. Let a closed-world play be *successful* when it contains a $\heartsuit_n$. Furthermore, for any closed-world behaviour $G \in \mathsf{G}_X$ and closed-world play $U \in \mathbb{W}(X)$, an *extension* of a state $\sigma \in G(U)$ to U' is a $\sigma' \in G(U')$ with $i \colon U \to U'$ and $G(i)(\sigma') = \sigma$. The extension σ' is *successful* when U' is. The intuition is that the behaviour G, before reaching U' with state σ', passed through U with state σ.

Definition 25 *The* fair *criterion $\perp\!\!\!\perp^f$ contains all closed-world behaviours G such that any state $\sigma \in G(U)$ for finite U admits a successful extension.*

Now call an extension of $\sigma \in G(U)$ *strict* when $U \to U'$ is not surjective, or, equivalently, when U' contains more moves than U. For any closed-world behaviour $G \in \mathsf{G}_X$, a state $\sigma \in G(U)$ is *G-maximal* when it has no strict extension.

Definition 26 *Let the* must *criterion $\perp\!\!\!\perp^m$ consist of all closed-world behaviours G such that for all closed-world U and G-maximal $\sigma \in G(U)$, U is successful.*

As explained in the introduction and Section 5.1, unlike in the standard setting, this definition of must testing equivalence distinguishes between the processes Ω and $\Omega|\overline{a}$. Indeed, take the CCS context $C = a.\heartsuit \mid \Box$, which we can implement by choosing as a test the strategy $T = [\![a.\heartsuit]\!]$ on a single player knowing one channel a. Taking I to consist of the sole channel a, the pushout Z as in Definition 23 consists of two players, say x for the observed strategy and y for the test strategy, sharing the channel a. Now, assuming that Ω loops deterministically, the global behaviour $G = \mathsf{GI}([[\![P]\!], T])$ has exactly one state on the identity play, and again exactly one state on the play π_1 consisting of only one fork move by x. Thus, G reaches a position with three players, say x_1 playing Ω, x_2 playing $\overline{a}$, and y playing $a.\heartsuit$. The play with infinitely many silent moves by x_1 is not maximal: we could insert (anywhere in the sequence of moves by x_1) a synchronisation move by x_2 and y, and then a tick move by the avatar of y. Essentially: our notion of play is more fair than just traces.

To get more intuition about must testing equivalence in our setting, we prove that it actually coincides with the testing equivalence generated by the following criterion:

Definition 27 *The* spatially fair *criterion $\perp\!\!\!\perp^{sf}$ contains all closed-world behaviours G such that any state $\sigma \in G(U)$ admits a successful extension.*

This criterion is almost like the fair criterion, except that we do not restrict to finite plays. The key result to show the equivalence is:

Theorem 4 *For any innocent strategy F on X, any state $\sigma \in \mathsf{GI}(F)(U)$ admits a $\mathsf{GI}(F)$-maximal extension.*

The proof is in Appendix B. Thanks to the theorem, we have:

Lemma 6 *For all $F \in \mathsf{S}_X$, $\mathsf{GI}(F) \in \perp\!\!\!\perp^m_X$ iff $\mathsf{GI}(F) \in \perp\!\!\!\perp^{sf}_X$.*

Proof: Let $G = \mathsf{GI}(F)$.

($\Rightarrow$) By Theorem 4, any state $\sigma \in G(U)$ has a G-maximal extension $\sigma' \in G(U')$, which is successful by hypothesis, hence σ has a successful extension.

($\Leftarrow$) Any G-maximal $\sigma \in G(U)$ admits by hypothesis a successful extension which may only be on U by G-maximality, and hence U is successful. $\Box$

(Note that U is not necessarily finite in the proof of the right-to-left implication, so that the argument does not apply to the fair criterion.)

Now comes the expected result:

Theorem 5 *For all $F, F' \in \mathsf{S}_X$, $F \sim_{\perp\!\!\!\perp^m} F'$ iff $F \sim_{\perp\!\!\!\perp^{sf}} F'$.*

Proof: *(⇒)* Consider two innocent strategies F and F' on X, and an innocent strategy G on Y (as in the pushout (8)). As in spatial decomposition (Proposition 5), copairing induces an isomorphism $\mathsf{S}_X \times \mathsf{S}_Y \to \mathsf{S}_Z$, and we have, using Lemma 6:

$$\begin{array}{rll} \mathsf{GI}[F,G] \in \perp\!\!\!\perp^{sf} & \text{iff} & \mathsf{GI}[F,G] \in \perp\!\!\!\perp^{m} \\ & \text{iff} & \mathsf{GI}[F',G] \in \perp\!\!\!\perp^{m} \\ & \text{iff} & \mathsf{GI}[F',G] \in \perp\!\!\!\perp^{sf} \end{array}$$

(⇐) Symmetric. □

Intuitively, must testing only considers spatially fair schedulings, in the sense that all players appearing in a play should be given the opportunity to play: no one should starve.

However, this is not the only source of unfairness, so that must testing and fair testing differ. To see this, consider the CCS process $P = \nu b.\mathtt{rec}\ x(a,b) := \overline{b}|(b.(x(a,b)) + \overline{a})\ \mathtt{in}\ x(a,b)$, that can repeatedly perform synchronisations on the private channel b, until it chooses to perform an output on a. We have $[\![\Omega]\!] \sim^{sf} [\![P]\!]$ while $[\![\Omega]\!] \not\sim^{f} [\![P]\!]$. Indeed, since the choice between doing a synchronisation on b or an output on a is done by a single player, the infinite play where the output on a is never performed is maximal: no player starve, we just have a player that repeatedly chooses the same branch, in an unfair way.

We leave for future work the investigation of such unfair scenarios and their correlation to the corresponding behaviours in classical presentations of CCS.

A Temporal Decomposition

This section is a proof of Theorem 2. Let us first review the general equivalences mentioned in the proof sketch. The product of a family of presheaf categories is isomorphic to the category of presheaves over the corresponding coproduct of categories:

Lemma 7 *We have $\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)} \cong [\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$.*

Furthermore, let the functor $\Delta \colon \mathsf{Set} \to \widehat{\mathbb{C}}$ map any set X to the constant presheaf mapping any $C \in \mathbb{C}$ to X. We have:

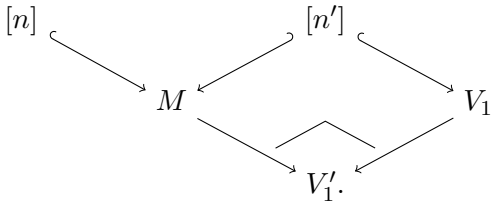
Lemma 8 *For any small category* $\mathbb{C}$, $\mathrm{Fam}(\widehat{\mathbb{C}}) \simeq (\widehat{\mathbb{C}} \downarrow \Delta)$.

Proof: A generalisation of the more well-known $\mathsf{Set}^X \simeq \mathsf{Set}/X$. □

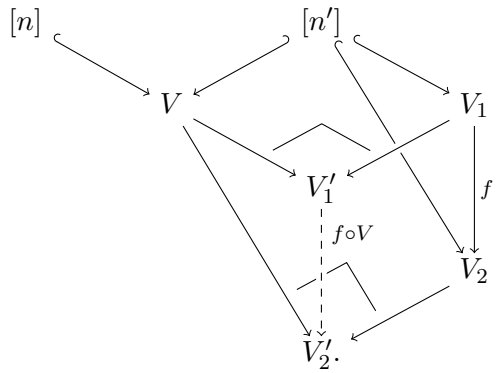
Corollary 1 *We have:*

$$\mathrm{Fam}\left(\prod_{M \in \mathcal{M}_n} \mathsf{S}_{\mathrm{cod}(M)}\right) \simeq ([\sum_{M \in \mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \downarrow \Delta).$$

We now construct the lax pushout (6). A first step is the construction, for each move $[n] \hookrightarrow M \hookleftarrow [n']$, of a functor $(- \circ M) \colon \mathbb{V}_{[n']} \to \mathbb{V}_{[n]}$ given by precomposition with M in $\mathsf{Cospan}(\widehat{\mathbb{C}})$. This functor maps any $V_1 \colon [n'] \hookrightarrow V_1$ to the view $V_1 \circ M$, i.e., the view $[n] \hookrightarrow V_1'$ defined by the colimit



This of course relies on the choice of such a colimit for every V and V_1. Any morphism $f \colon V_1 \to V_2$ in $\mathbb{V}_{[n']}$, letting $V_2' = V_2 \circ V$, is mapped to the dashed morphism induced by universal property of pushout in



Once the choice has been made on objects, the map for morphisms is determined uniquely.

This family of functors allows us to decompose $\mathbb{V}_{[n]}$ as follows:

Lemma 9 *The diagram*

$$\begin{array}{ccc} \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} & = & \sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)} \\ {\scriptstyle !}\downarrow & \overset{\lambda}{\swarrow} & \downarrow{\scriptstyle [-\circ M]_{M\in\mathcal{M}_n}} \\ 1 & \xrightarrow[\ulcorner id_{[n]} \urcorner]{} & \mathbb{V}^{op}_{[n]} \end{array} \tag{9}$$

is a lax pushout, where $\lambda_{M,V} \colon id_{[n]} \to M \circ V$*, seen in* $\mathbb{V}_{[n]}$*, is the obvious inclusion.*

Proof: For any category $\mathbb{C}$, taking such a lax pushout of $id_{\mathbb{C}}$ with 1 just adds a terminal object to $\mathbb{C}$. The rest is an easy verification. A dual result of course holds with $\mathbb{V}_{[n]}$, reversing the direction of λ. □

Now, it is well-known that, in any small 2-category $\mathbb{K}$, any contravariant hom-2-functor, i.e., 2-functor of the shape $\mathbb{K}(-, X)$ for $X \in K$, maps weighted colimits in $\mathbb{K}$ to weighted limits in Cat. For an introduction to weighted limits and colimits in the case of enrichment over Cat, see Kelly [26]. Here, for any 2-category P, and 2-functors $G \colon P \to \mathbb{K}$ and $J \colon P^{op} \to \mathsf{Cat}$, any colimit $L = J \star G$ of G weighted by J with unit $\xi \colon J \to \mathbb{K}(G(-), L)$ in $[P^{op}, \mathsf{Cat}]$ is mapped, for any object $X \in \mathbb{K}$, by the hom-2-functor $\mathbb{K}(-, X)$ to a limit of $\mathbb{K}(G(-), X) \colon P^{op} \to \mathsf{Cat}$ weighted by J in Cat, with unit $\mathbb{K}(\xi, X) \colon J \to \mathsf{Cat}(\mathbb{K}(L, X), \mathbb{K}(G(-), X))$, in Cat. In particular, lax pushouts are mapped to lax pullbacks. As usual, considering a larger universe, we may replace Cat with CAT and obtain the same results with $\mathbb{K} = \mathsf{Cat}$.

Recalling our lax pushout (9) and taking the hom-categories to Set, we obtain a lax pullback

$$\begin{array}{ccc} [\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] & = & [\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \\ {\scriptstyle !^*}\uparrow & \overset{\lambda^*}{\swarrow} & \uparrow \\ \mathsf{Set} & \longleftarrow & \mathsf{S}_{[n]} \end{array}$$

in CAT, i.e., a comma category. But observe that restriction along ! is precisely $\Delta \colon \mathsf{Set} \to [\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}]$, so we have indeed shown that $\mathsf{S}_{[n]}$ is a comma category $[\sum_{M\in\mathcal{M}_n} \mathbb{V}^{op}_{\mathrm{cod}(M)}, \mathsf{Set}] \downarrow \Delta$.

B Maximal Extensions

This section is a proof of Theorem 4.

Lemma 10 *For any position X, the category $\mathbb{W}(X)$ of closed-world plays is a preorder.*

Proof: Easy. □

In the following, we consider the quotient poset.

Lemma 11 *In $\mathbb{W}(X)$, any non-decreasing chain admits an upper bound.*

Recall $\mathcal{M}$, the graph of all basic moves, and the set $\mathcal{M}_n$ of edges from n, for each n. Let now, for each n, $\mathcal{M}_n^f$ be the analogous set with full moves, i.e., the set of isomorphism classes of full moves from $[n]$.

Lemma 12 *For each play $U \in \mathbb{E}_X$, the coproduct of all s maps from full moves*

$$\left(\sum_{n \in \mathsf{FinOrd}} \sum_{M \in \mathcal{M}_n^f} U(M) \right) \to \sum_{n \in \mathsf{FinOrd}} U[n], \tag{10}$$

is injective.

Recall here that for forking, we have also called s the common composite $l \circ s = r \circ s$ (see the discussion following Definition 3).

Proof: By induction on U. □

Lemma 13 *Any non-decreasing sequence in the poset $\mathbb{W}(X)$ admits its colimit in $\widehat{\mathbb{C}}$ as an upper bound.*

Proof: Consider any increasing sequence $U^1 \hookrightarrow U^2 \hookrightarrow \ldots$ of plays in $\mathbb{W}(X)$. Let U be its colimit in $\widehat{\mathbb{C}}$. We want to prove that U is a play.

First, observe that U satisfies joint injectivity of s-maps as in Lemma 12: indeed, if we had a player p and two full moves M and M' such that $s(M) = s(M') = p$, then all of M, M', and p would appear in some U^i, which, being a play, has to satisfy joint injectivity.

For each n, U^n comes with a sequence of compatible (closed-world) extended moves

$$X = X_0^n \hookrightarrow M_1^n \hookleftarrow X_1^n \hookrightarrow \ldots \hookleftarrow X_{i-1}^n \hookrightarrow M_i^n \hookleftarrow X_i^n \hookrightarrow \ldots$$

which are also (by the colimit cocone) morphisms over U in $\widehat{\mathbb{C}}$. For each $i \geq 1$, taking the colimit of the i first moves yields a finite play $X \hookrightarrow U_i^n \hookleftarrow X_i^n$. By convention, letting $U_0^n = X$ extends this to $i \geq 0$. Similarly, we may

consider all the given plays infinite, by accepting not only extended moves, but also identity cospans.

We consider the poset of pairs $(N,n) \in \{(0,0)\} \uplus \sum_{N \in \mathsf{FinOrd}^*} N$, with lexicographic order, i.e., $(N,n) \leq (N',n')$ when $N < N'$ or when $N = N'$ and $n \leq n'$.

We will construct by induction on (N,n) a sequence of composable closed-world moves, with colimit U', such that for all (N,n), $U^n_{N-n+1} \subseteq U'$ in $\mathbb{W}(X)/U$. More precisely, we construct for each (N,n) an integer $K_{N,n}$ and a sequence

$$X = X^{N,n}_0 \hookrightarrow M^{N,n}_1 \hookleftarrow X^{N,n}_1 \hookrightarrow \ldots \hookleftarrow X^{N,n}_{K_{N,n}-1} \hookrightarrow M^{N,n}_{K_{N,n}} \hookleftarrow X^{N,n}_{K_{N,n}},$$

(again, if $K_{N,n} = 0$, we mean the empty sequence) such that

- for all $(N',n') < (N,n)$, we have $K_{N',n'} \leq K_{N,n}$ and the sequence $(M^{N',n'}_i)_{i \in K_{N',n'}}$ is a prefix of $(M^{N,n}_{i \in K_{N,n}})$;
- and the colimit, say $U_{N,n}$, of $(M^{N,n}_i)_{i \in K_{N,n}}$ is such that for all $(N',n') \leq (N,n)$, $U^{n'}_{N-n'+1} \subseteq U_{N,n}$ in $\mathbb{W}(X)/U$.

For the base case, we let $K_{0,0} = 0$, which forces $M^{0,0}$ to be the empty sequence on X.

For the induction step, consider any $(N,n) \neq (0,0)$, and let (N_0,n_0) be the predecessor of (N,n). The induction hypothesis gives a K_{N_0,n_0} and a sequence $(M^{N_0,n_0}_i)_{i \in K_{N_0,n_0}}$ satisfying some hypotheses, among which the existence of a diagram

$$\begin{array}{ccccccc} X & \longrightarrow & U^n_{N-n} & \longleftarrow & X^n_{N-n} & \longrightarrow & M^n_{N-n+1} \longleftarrow X^n_{N-n+1} \\ \| & & \downarrow & & & & \\ X & \longrightarrow & U_{N_0,n_0} & \longleftarrow & X^{N_0,n_0}_{K_{N_0,n_0}} & & \end{array}$$

over U.

Now, if $M^n_{N-n+1} \to U$ factors through U_{N_0,n_0}, then we put $K_{N,n} = K_{N_0,n_0}$ and $(M^{N,n}_i)_{i \in K_{N,n}} = (M^{N_0,n_0}_i)_{i \in K_{N_0,n_0}}$, and all induction hypotheses go through.

Otherwise, M^n_{N-n+1} is played by players in X^n_{N-n} which are not in the joint image of all s maps (10) in U_{N_0,n_0}, otherwise s maps in U could not be jointly injective, contradicting Lemma 12. Technically, the diagram

$$X^n_{N-n} \to M^n_{N-n+1} \leftarrow X^n_{N-n+1}$$

is obtained by pushing some (non-extended) closed-world move $Y \to M \leftarrow Y'$ along some morphism $I \to Z$ from an interface I, and the induced morphism $Y \to X^n_{N-n} \to U^n_{N-n} \to U_{N_0,n_0}$ factors through $X^{N_0,n_0}_{K_{N_0,n_0}}$. We consider the subposition $Z' \subseteq X^{N_0,n_0}_{K_{N_0,n_0}}$ making

$$\begin{array}{ccc} I & \hookrightarrow & Y \\ \downarrow & & \downarrow \\ Z' & \hookrightarrow & X^{N_0,n_0}_{K_{N_0,n_0}} \end{array}$$

a pushout; Z' consists of the players in $X^{N_0,n_0}_{K_{N_0,n_0}}$ that are not in the image of Y, plus their names, plus possibly missing names from I.

Then, pushing $Y \to M \leftarrow Y'$ along $I \to Z'$, we obtain an extended move $X^{N_0,n_0}_{K_{N_0,n_0}} \hookrightarrow M' \hookleftarrow X'$. We let $K_{N,n} = K_{N_0,n_0} + 1$ and define $(M^{N,n}_i)_{i \in K_{N,n}}$ to be the extension of $(M^{N_0,n_0}_i)_{i \in K_{N_0,n_0}}$ by M'. This induces a unique map $U_{N,n} \to U$ by universal property of $U_{N,n}$ as a colimit. All induction hypotheses go through; in particular, U^n_{N-n+1} is a union $U^n_{N-n} \cup M^n_{N-n+1}$ in $\mathbb{W}(X)/U$, and actually a union $U^n_{N-n} \cup M$; similarly, $U_{N,n} = U_{N_0,n_0} \cup M$; so, since we have $U^n_{N-n} \subseteq U_{N_0,n_0}$ by induction hypothesis, we obtain $U^n_{N-n+1} \subseteq U_{N,n}$.

The sequences $M^{N,n}$ induce by union a possibly infinite sequence of closed-world extended moves, i.e., a closed-world play U', such that for all (N,n), $U^n_{N-n+1} \subseteq U'$, hence, for all n, $U^n \subseteq U' \subseteq U$, i.e., $U' \cong U$. Thus, U is indeed a play. □

We are almost ready for proving Theorem 4. We just need one more lemma. Consider any innocent strategy F on X, play $U \in \mathbb{W}(X)$, and any state $\sigma \in \mathsf{GI}(F)(U)$. Consider now the poset F_σ of $\mathsf{GI}(F)$-extensions of σ (made into a poset by choosing a skeleton of $\mathbb{W}(X)$), where $\sigma' \in F(U') \leq \sigma'' \in F(U'')$ iff $U' \leq U''$. This poset is not empty, since it contains σ. Furthermore, we have:

Lemma 14 *Any non-decreasing sequence in F_σ admits an upper bound.*

Proof: Any such sequence, say $(\sigma_i)_{i \in \mathsf{FinOrd}}$, induces a non-decreasing sequence of plays in $\mathbb{W}(X)$, say $(U_i)_i$, which by Lemma 13 admits its colimit, say U', as an upper bound. Now, any view inclusion $j \colon V \hookrightarrow U'$, factors through some U_i, and we let $\sigma_j = (\sigma_i)_{|V}$ (this does not depend on the choice of i). This assignment determines (by innocence of F and by construction of the right Kan extension as an end) an element $\sigma' \in F(U')$, which is an upper bound for $(\sigma_i)_{i \in \mathsf{FinOrd}}$. □

Proof of Theorem 4: Consider any innocent strategy F on X, play $U \in \mathbb{W}(X)$, and any state $\sigma \in \mathsf{Gl}(F)(U)$. Consider as above the poset F_σ of $\mathsf{Gl}(F)$-extensions of σ. By the last lemma, we may apply Zorn's lemma to choose a maximal element of F_σ, which is a $\mathsf{Gl}(F)$-maximal extension of σ. □

References

[1] Emmanuel Beffara. *Logique, réalisabilité et concurrence*. PhD thesis, Université Paris 7, December 2005.

[2] Marcello M. Bonsangue, Jan J. M. M. Rutten, and Alexandra Silva. A Kleene theorem for polynomial coalgebras. In Luca de Alfaro, editor, *FOSSACS*, volume 5504 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2009.

[3] Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In Insup Lee and Scott A. Smolka, editors, *CONCUR*, volume 962 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 1995.

[4] Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical Computer Science*, 115(1):43–62, 1993.

[5] Aurelio Carboni and Peter Johnstone. Connected limits, familial representability and artin glueing. *Mathematical Structures in Computer Science*, 5(4):441–459, 1995.

[6] Aurelio Carboni and Peter Johnstone. Corrigenda for 'connected limits, familial representability and artin glueing'. *Mathematical Structures in Computer Science*, 14(1):185–187, 2004.

[7] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theor. Comput. Sci.*, 34:83–133, 1984.

[8] Olivier Delande and Dale Miller. A neutral approach to proof and refutation in mall. In *LICS '08* [30], pages 498–508.

[9] H. Ehrig, H.-J. Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[10] Marcelo P. Fiore. Second-order and dependently-sorted abstract syntax. In *LICS '08* [30], pages 57–68.

[11] Fabio Gadducci, Reiko Heckel, and Mercè Llabrés. A bi-categorical axiomatisation of concurrent graph rewriting. *Electronic Notes in Theoretical Computer Science*, 29, 1999.

[12] Fabio Gadducci and Ugo Montanari. The tile model. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 133–166. The MIT Press, 2000.

[13] Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[14] Yves Guiraud and Philippe Malbos. Higher-dimensional categories with finite derivation type. *Theory and Applications of Categories*, 22(18):420–278, 2009.

[15] André Hirschowitz and Marco Maggesi. Modules over monads and linearity. In Daniel Leivant and Ruy J. G. B. de Queiroz, editors, *WoLLIC*, volume 4576 of *Lecture Notes in Computer Science*, pages 218–237. Springer, 2007.

[16] André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Information and Computation*, 208(5):545–564, 2010.

[17] André, Michel, and Tom Hirschowitz. Contraction-free proofs and finitary games for linear logic. *Electronic Notes in Theoretical Computer Science*, 249:287–305, 2009.

[18] Tom Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. Preprint. http://hal.archives-ouvertes.fr/hal-00540205/en/, 2010.

[19] Tom Hirschowitz and Damien Pous. Innocent strategies as presheaves and interactive equivalences for CCS. In Alexandra Silva, Simon Bliudze, Roberto Bruni, and Marco Carbone, editors, *ICE*, volume 59 of *EPTCS*, pages 2–24, 2011.

[20] Martin Hyland. *Semantics and Logics of Computation*, chapter Game Semantics. Cambridge University Press, 1997.

[21] Bart Jacobs. *Categorical Logic and Type Theory*. Number 141 in Studies in Logic and the Foundations of Mathematics. North Holland, Amsterdam, 1999.

[22] Ole H. Jensen and Robin Milner. Bigraphs and mobile processes (revised). Technical Report TR580, University of Cambridge, 2004.

[23] P. T. Johnstone, S. Lack, and P. Sobociński. Quasitoposes, quasiadhesive categories and Artin glueing. In *CALCO*, volume 4624 of *LNCS*, pages 312–326. Springer Verlag, 2007.

[24] André Joyal, Mogens Nielsen, and Glynn Winskel. Bisimulation and open maps. In *LICS '93*, pages 418–427. IEEE Computer Society, 1993.

[25] Stefano Kasangian and Anna Labella. Observational trees as models for concurrency. *Mathematical Structures in Computer Science*, 9(6):687–718, 1999.

[26] G. M. Kelly. Elementary observations on 2-categorical limits. *Bulletin of the Australian Mathematical Society*, 39:301–317, 1989.

[27] Joachim Kock. Polynomial functors and trees. *International Mathematics Research Notices*, 2011(3):609–673, 2011.

[28] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theor. Comput. Sci.*, 308(1-3):259–276, 2003.

[29] James J. Leifer and Robin Milner. Deriving bisimulation congruences for reactive systems. In Catuscia Palamidessi, editor, *CONCUR*, volume 1877 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2000.

[30] *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society, 2008.

[31] Saunders Mac Lane. *Categories for the Working Mathematician*. Number 5 in Graduate Texts in Mathematics. Springer, 2nd edition, 1998.

[32] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer, 1992.

[33] Paul-André Melliès. Asynchronous games 2: the true concurrency of innocence. In *Proc. CONCUR '04*, volume 3170 of *LNCS*, pages 448–465. Springer Verlag, 2004.

[34] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[35] V. Natarajan and Rance Cleaveland. Divergence and fair testing. In Zoltán Fülöp and Ferenc Gécseg, editors, *ICALP*, volume 944 of *Lecture Notes in Computer Science*, pages 648–659. Springer, 1995.

[36] Tobias Nipkow. Higher-order critical pairs. In *LICS '91*, pages 342–349. IEEE Computer Society, 1991.

[37] Gordon D. Plotkin. A structural approach to operational semantics. DAIMI Report FN-19, Computer Science Department, Aarhus University, 1981.

[38] Julian Rathke and Pawel Sobocinski. Deconstructing behavioural theories of mobility. In *IFIP TCS*, volume 273 of *IFIP*, pages 507–520. Springer, 2008.

[39] Vladimiro Sassone and Pawel Sobociński. Deriving bisimulation congruences using 2-categories. *Nordic Journal of Computing*, 10(2), 2003.

[40] Peter Sewell. From rewrite to bisimulation congruences. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR*, volume 1466 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 1998.

[41] Daniele Turi and Gordon D. Plotkin. Towards a mathematical operational semantics. In *LICS '97*, pages 280–291, 1997.

[42] Rob J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In Jos C. M. Baeten and Jan Willem Klop, editors, *CONCUR*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

[43] Angelo Vistoli. Notes on Grothendieck topologies, fibered categories and descent theory. Preprint. http://arxiv.org/abs/math/0412512, 2007.

© Scientific Annals of Computer Science 2012

hal-00639716, version 5 - 11 Jul 2012

Checking NFA equivalence with bisimulations up to congruence

Filippo Bonchi Damien Pous
CNRS, ENS Lyon, Université de Lyon, LIP (UMR 5668)
{filippo.bonchi,damien.pous}@ens-lyon.fr

Abstract

We introduce *bisimulation up to congruence* as a technique for proving language equivalence of non-deterministic finite automata. Exploiting this technique, we devise an optimisation of the classical algorithm by Hopcroft and Karp [16]. We compare our approach to the recently introduced antichain algorithms, by analysing and relating the two underlying coinductive proof methods. We give concrete examples where we exponentially improve over antichains; experimental results moreover show non negligible improvements.

Keywords Language Equivalence, Automata, Bisimulation, Coinduction, Up-to techniques, Congruence, Antichains.

1. Introduction

Checking language equivalence of finite automata is a classical problem in computer science, which finds applications in many fields ranging from compiler construction to model checking.

Equivalence of deterministic finite automata (DFA) can be checked either via minimisation [9, 15] or through Hopcroft and Karp's algorithm [2, 16], which exploits an instance of what is nowadays called a *coinduction proof principle* [24, 27, 29]: two states recognise the same language if and only if there exists a *bisimulation* relating them. In order to check the equivalence of two given states, Hopcroft and Karp's algorithm creates a relation containing them and tries to build a bisimulation by adding pairs of states to this relation: if it succeeds then the two states are equivalent, otherwise they are different.

On the one hand, minimisation algorithms have the advantage of checking the equivalence of all the states at once (while Hopcroft and Karp's algorithm only check a given pair of states). On the other hand, they have the disadvantage of needing the whole automata from the beginning[1], while Hopcroft and Karp's algorithm can be executed "on-the-fly" [12], on a lazy DFA whose transitions are computed on demand.

This difference is fundamental for our work and for other recently introduced algorithms based on *antichains* [1, 33]. Indeed, when starting from non-deterministic finite automata (NFA), the powerset construction used to get deterministic automata induces an exponential factor. In contrast, the algorithm we introduce in this work for checking equivalence of NFA (as well as those in [1, 33]) usually does not build the whole deterministic automaton, but just a small part of it. We write "usually" because in few bad cases, the algorithm still needs exponentially many states of the DFA.

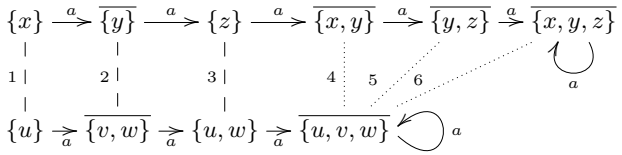
Our algorithm is grounded on a simple observation on determinised NFA: for all sets X and Y of states of the original NFA, the union (written $+$) of the language recognised by X (written $[\![X]\!]$) and the language recognised by Y ($[\![Y]\!]$) is equal to the language recognised by the union of X and Y ($[\![X+Y]\!]$). In symbols:

$$[\![X+Y]\!] = [\![X]\!] + [\![Y]\!] \tag{1}$$

This fact leads us to introduce a sound and complete proof technique for language equivalence, namely *bisimulation up to context*, that exploits both *induction* (on the operator $+$) and *coinduction*: if a bisimulation R equates both the (sets of) states X_1, Y_1 and X_2, Y_2, then $[\![X_1]\!] = [\![Y_1]\!]$ and $[\![X_2]\!] = [\![Y_2]\!]$ and, by (1), we can immediately conclude that also $X_1 + X_2$ and $Y_1 + Y_2$ are language equivalent. Intuitively, bisimulations up to context are bisimulations which *do not need to relate* $X_1 + X_2$ and $Y_1 + Y_2$ when X_1 (resp. X_2) and Y_1 (resp. Y_2) are already related.

To illustrate this idea, let us check the equivalence of states x and u in the following NFA. (Final states are overlined, labelled edges represent transitions.)

The determinised automaton is depicted below.



Each state is a set of states of the NFA, final states are overlined: they contain at least one final state of the NFA. The numbered lines show a relation which is a bisimulation containing x and u. Actually, this is the relation that is built by Hopcroft and Karp's algorithm (the numbers express the order in which pairs are added).

The dashed lines (numbered by 1, 2, 3) form a smaller relation which is not a bisimulation, but a bisimulation up to context: the equivalence of states $\{x, y\}$ and $\{u, v, w\}$ could be immediately deduced from the fact that $\{x\}$ is related to $\{u\}$ and $\{y\}$ to $\{v, w\}$, without the need of further exploring the determinised automaton.

Bisimulations up-to, and in particular bisimulations up to context, have been introduced in the setting of concurrency theory [24,

[1] There are few exceptions, like [19] which minimises labelled transition systems w.r.t. bisimilarity rather than trace equivalence.

[Copyright notice will appear here once 'preprint' option is removed.]

hal-00639716, version 5 - 11 Jul 2012

25, 28] as a proof technique for bisimilarity of CCS or π-calculus processes. As far as we know, they have never been used for proving language equivalence of NFA.

Among these techniques one should also mention *bisimulation up to equivalence*, which, as we show in this paper, is implicitly used in the original Hopcroft and Karp's algorithm. This technique can be briefly explained by noting that not all bisimulations are equivalence relations: it might be the case that a bisimulation relates (for instance) X and Y, Y and Z but not X and Z. However, since $[\![X]\!] = [\![Y]\!]$ and $[\![Y]\!] = [\![Z]\!]$, we can immediately conclude that X and Z recognise the same language. Analogously to bisimulations up to context, a bisimulation up to equivalence *does not need to relate* X and Z when they are both related to some Y.

The techniques of up-to equivalence and up-to context can be combined resulting in a powerful proof technique which we call *bisimulation up to congruence*. Our algorithm is in fact just an extension of Hopcroft and Karp's algorithm that attempts to build a bisimulation up to congruence instead of a bisimulation up to equivalence. An important consequence, when using up to congruence, is that we do not need to build the whole deterministic automata, but just those states that are needed for the bisimulation up-to. For instance, in the above NFA, the algorithm stops after equating z and $u + v$ and does not build the remaining four states. Despite their use of the up to equivalence technique, this is not the case with Hopcroft and Karp's algorithm, where all accessible subsets of the deterministic automata have to be visited at least once.

The ability of visiting only a small portion of the determinised automaton is also the key feature of the antichain algorithm [33] and its optimisation exploiting similarity [1]. The two algorithms are designed to check *language inclusion* rather than equivalence, but we can relate these approaches by observing that the two problems are equivalent ($[\![X]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$ and $[\![Y]\!] \subseteq [\![X]\!]$; and $[\![X]\!] \subseteq [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X + Y]\!] = [\![Y]\!]$).

In order to compare with these algorithms, we make explicit the coinductive up-to technique underlying the antichain algorithm [33]. We prove that this technique can be seen as a restriction of up to congruence, for which *symmetry* and *transitivity* are not allowed. As a consequence, the antichain algorithm usually needs to explore more states than our algorithm. Moreover, we show how to integrate the optimisation proposed in [1] in our setting, resulting in an even more efficient algorithm.

Summarising, the contributions of this work are

(1) the observation that Hopcroft and Karp implicitly use bisimulations up to equivalence (Section 2),

(2) an efficient algorithm for checking language equivalence (and inclusion), based on a powerful up to technique (Section 3),

(3) a comparison with antichain algorithms, by recasting them into our coinductive framework (Sections 4 and 5).

Outline

Section 2 recalls Hopcroft and Karp's algorithm for DFA, showing that it implicitly exploits bisimulation up to equivalence. Section 3 describes the novel algorithm, based on bisimulations up to congruence. We compare this algorithm with the antichain one in Section 4, and we show how to exploit similarity in Section 5. Section 6 is devoted to benchmarks. Sections 7 and 8 discuss related and future works. Omitted proofs can be found in the Appendix.

Notation

We denote sets by capital letters $X, Y, S, T \ldots$ and functions by lower case letters $f, g, \ldots$ Given sets X and Y, $X \times Y$ is their Cartesian product, $X \uplus Y$ is the disjoint union and X^Y is the set of functions $f\colon Y \to X$. Finite iterations of a function $f\colon X \to X$ are denoted by f^n (formally, $f^0(x) = x$, $f^{n+1}(x) = f(f^n(x))$). The collection of subsets of X is denoted by $\mathcal{P}(X)$. The (omega) iteration of a function $f\colon \mathcal{P}(X) \to \mathcal{P}(X)$ is denoted by f^ω (formally, $f^\omega(Y) = \bigcup_{n \geq 0} f^n(Y)$). For a set of letters A, $A^\star$ denotes the set of all finite words over A; ϵ the empty word; and $w_1 w_2$ the concatenation of words $w_1, w_2 \in A^\star$. We use 2 for the set $\{0, 1\}$ and $2^{A^\star}$ for the set of all languages over A.

2. Hopcroft and Karp's algorithm for DFA

A deterministic finite automaton (DFA) over the alphabet A is a triple (S, o, t), where S is a finite set of states, $o\colon S \to 2$ is the output function, which determines if a state $x \in S$ is final ($o(x) = 1$) or not ($o(x) = 0$), and $t\colon S \to S^A$ is the transition function which returns, for each state x and for each letter $a \in A$, the next state $t_a(x)$. For $a \in A$, we write $x \xrightarrow{a} x'$ to mean that $t_a(x) = x'$. For $w \in A^\star$, we write $x \xrightarrow{w} x'$ for the least relation such that (1) $x \xrightarrow{\epsilon} x$ and (2) $x \xrightarrow{aw'} x'$ iff $x \xrightarrow{a} x''$ and $x'' \xrightarrow{w'} x'$.

For any DFA, there exists a function $[\![-]\!]\colon S \to 2^{A^\star}$ mapping states to languages, defined for all $x \in S$ as follows:

$$[\![x]\!](\epsilon) = o(x) \ , \qquad [\![x]\!](aw) = [\![t_a(x)]\!](w) \ .$$

The language $[\![x]\!]$ is called the language accepted by x. Given two automata (S_1, o_1, t_1) and (S_2, o_2, t_2), the states $x_1 \in S_1$ and $x_2 \in S_2$ are said to be *language equivalent* (written $x_1 \sim x_2$) iff they accept they same language.

Remark 1. *In the following, we will always consider the problem of checking the equivalence of states of one single and fixed automaton (S, o, t). We do not loose generality since for any two automata (S_1, o_1, t_1) and (S_2, o_2, t_2) it is always possible to build an automaton $(S_1 \uplus S_2, o_1 \uplus o_2, t_1 \uplus t_2)$ such that the language accepted by every state $x \in S_1 \uplus S_2$ is the same as the language accepted by x in the original automaton (S_i, o_i, t_i). For this reason, we also work with automata without explicit initial states: we focus on the equivalence of two arbitrary states of a fixed DFA.*

2.1 Proving language equivalence via coinduction

We first define bisimulation. We make explicit the underlying notion of progression which we need in the sequel.

Definition 1 (Progression, Bisimulation)**.** *Given two relations $R, R' \subseteq S \times S$ on states, R* progresses to *R', denoted $R \rightarrowtail R'$, if whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x)\ R'\ t_a(y)$.*

A bisimulation *is a relation R such that $R \rightarrowtail R$.*

As expected, bisimulation is a sound and complete proof technique for checking language equivalence of DFA:

Proposition 1 (Coinduction)**.** *Two states are language equivalent iff there exists a bisimulation that relates them.*

2.2 Naive algorithm

Figure 1 shows a naive version of Hopcroft and Karp's algorithm for checking language equivalence of the states x and y of a deterministic finite automaton (S, o, t). Starting from x and y, the algorithm builds a relation R that, in case of success, is a bisimulation. In order to do that, it employs the set (of pairs of states) $todo$ which, intuitively, at any step of the execution, contains the pairs (x', y') that must be checked: if (x', y') already belongs to R, then it has already been checked and nothing else should be done. Otherwise, the algorithm checks if x' and y' have the same outputs (i.e., if both are final or not). If $o(x') \neq o(y')$, then x and y are different.

hal-00639716, version 5 - 11 Jul 2012

$\underline{\texttt{Naive}(x,y)}$

```
(1) R is empty; todo is empty;
(2) insert (x,y) in todo;
(3) while todo is not empty, do {
 (3.1)  extract (x',y') from todo;
 (3.2)  if (x',y') ∈ R then skip;
 (3.3)  if o(x') ≠ o(y') then return false;
 (3.4)  for all a ∈ A,
          insert (t_a(x'), t_a(y')) in todo;
 (3.5)  insert (x',y') in R;
(4) return true;
```

Figure 1. Naive algorithm for checking the equivalence of states x and y of a DFA (S, o, t); R and $todo$ are sets of pairs of states. The code of $\texttt{HK}(x,y)$ is obtained by replacing step `3.2` with `if` $(x',y') \in e(R)$ `then skip`.

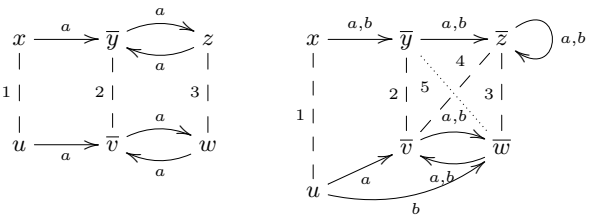


Figure 2. Checking for DFA equivalence.

If $o(x') = o(y')$, then the algorithm inserts (x', y') in R and, for all $a \in A$, the pairs $(t_a(x'), t_a(y'))$ in $todo$.

Proposition 2. *For all $x, y \in S$, $x \sim y$ iff* $\texttt{Naive}(x,y)$.

Proof. We first observe that if $\texttt{Naive}(x,y)$ returns true then the relation R that is built before arriving to step `4` is a bisimulation. Indeed, the following proposition is an invariant for the loop corresponding to step `3`:

$$R \rightarrowtail R \cup todo$$

This invariant is preserved since at any iteration of the algorithm, a pair (x', y') is removed from $todo$ and inserted in R after checking that $o(x') = o(y')$ and adding $(t_a(x'), t_a(y'))$ for all $a \in A$ in $todo$. Since $todo$ is empty at the end of the loop, we eventually have $R \rightarrowtail R$, i.e., R is a bisimulation. By Proposition 1, $x \sim y$.

We now prove that if $\texttt{Naive}(x,y)$ returns false, then $x \not\sim y$. Note that for all (x', y') inserted in $todo$, there exists a word $w \in A^\star$ such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Since $o(x') \neq o(y')$, then $[\![x']\!](\epsilon) \neq [\![y']\!](\epsilon)$ and thus $[\![x]\!](w) = [\![x']\!](\epsilon) \neq [\![y']\!](\epsilon) = [\![y]\!](w)$, that is $x \not\sim y$. □

Since both Hopcroft and Karp's algorithm and the one we introduce in Section 3 are simple variations of this naive one, it is important to illustrate its execution with an example. Consider the DFA with input alphabet $A = \{a\}$ in the left-hand side of Figure 2, and suppose we want to check that x and u are language equivalent.

During the initialisation, (x, u) is inserted in $todo$. At the first iteration, since $o(x) = 0 = o(u)$, (x, u) is inserted in R and (y, v) in $todo$. At the second iteration, since $o(y) = 1 = o(v)$, (y, v) is inserted in R and (z, w) in $todo$. At the third iteration, since $o(z) = 0 = o(w)$, (z, w) is inserted in R and (y, v) in $todo$. At the fourth iteration, since (y, v) is already in R, the algorithm does nothing. Since there are no more pairs to check in $todo$, the relation R is a bisimulation and the algorithm terminates returning true.

These iterations are concisely described by the numbered dashed lines in Figure 2. The line i means that the connected pair is inserted in R at iteration i. (In the sequel, when enumerating iterations, we ignore those where a pair from $todo$ is already in R so that there is nothing to do.)

Remark 2. *Unless it finds a counter-example,* `Naive` *constructs the* smallest *bisimulation that relates the two starting states (see Proposition 8 in Appendix A). On the contrary, minimisation algorithms [9, 15] are designed to compute the* largest *bisimulation relation for a given automaton. For instance, taking automaton on the left of Figure 2, they would equate the states x and w which are language equivalent, while $\texttt{Naive}(x,u)$ does not relate them.*

2.3 Hopcroft and Karp's algorithm

The naive algorithm is quadratic: a new pair is added to R at each non-trivial iteration, and there are only n^2 such pairs, where $n = |S|$ is the number of states of the DFA. To make this algorithm (almost) linear, Hopcroft and Karp actually record a set of *equivalence classes* rather than a set of visited pairs. As a consequence, their algorithm may stop earlier, when an encountered pair of states is not already in R but in its reflexive, symmetric, and transitive closure. For instance in the right-hand side example from Figure 2, we can stop when we encounter the dotted pair (y, w), since these two states already belong to the same equivalence class according to the four previous pairs.

With this optimisation, the produced relation R contains at most n pairs (two equivalence classes are merged each time a pair is added). Formally, and ignoring the concrete data structure to store equivalence classes, Hopcroft and Karp's algorithm consists in simply replacing step `3.2` in Figure 1 with

```
(3.2)  if (x',y') ∈ e(R) then skip;
```

where $e \colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is the function mapping each relation $R \subseteq S \times S$ into its symmetric, reflexive, and transitive closure. We hereafter refer to this algorithm as `HK`.

2.4 Bisimulations up-to

We now show that the optimisation used by Hopcroft and Karp corresponds to exploiting an "up-to technique".

Definition 2 (Bisimulation up-to)**.** *Let $f \colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ be a function on relations on S. A relation R is a* bisimulation up to f *if $R \rightarrowtail f(R)$, i.e., whenever $x\ R\ y$ then*

1. *$o(x) = o(y)$ and*
2. *for all $a \in A$, $t_a(x)\ f(R)\ t_a(y)$.*

With this definition, Hopcroft and Karp's algorithm just consists in trying to build a bisimulation up to e. To prove the correctness of the algorithm it suffices to show that any bisimulation up to e is contained in a bisimulation. We use for that the notion of compatible function [26, 28]:

Definition 3 (Compatible function)**.** *A function $f \colon \mathcal{P}(S \times S) \to \mathcal{P}(S \times S)$ is* compatible *if it is monotone and it preserves progressions: for all $R, R' \subseteq S \times S$,*

$$R \rightarrowtail R' \text{ entails } f(R) \rightarrowtail f(R')\,.$$

Proposition 3. *Let f be a compatible function. Any bisimulation up to f is contained in a bisimulation.*

Proof. Suppose that R is a bisimulation up to f, i.e., that $R \rightarrowtail f(R)$. Using compatibility of f and by a simple induction on n, we get $\forall n,\ f^n(R) \rightarrowtail f^{n+1}(R)$. Therefore, we have

$$\bigcup_n f^n(R) \rightarrowtail \bigcup_n f^n(R)\,,$$

hal-00639716, version 5 - 11 Jul 2012

in other words, $f^\omega(R) = \bigcup_n f^n(R)$ is a bisimulation. This latter relation trivially contains R, by taking $n = 0$. □

We could prove directly that e is a compatible function; we however take a detour to ease our correctness proof for the algorithm we propose in Section 3.

Lemma 1. *The following functions are compatible:*

id: *the identity function;*
$f \circ g$: *the composition of compatible functions f and g;*
$\bigcup F$: *the pointwise union of an arbitrary family F of compatible functions: $\bigcup F(R) = \bigcup_{f \in F} f(R)$;*
f^ω: *the (omega) iteration of a compatible function f.*

Lemma 2. *The following functions are compatible:*

- *the constant reflexive function: $r(R) = \{(x, x) \mid \forall x \in S\}$;*
- *the converse function: $s(R) = \{(y, x) \mid x\ R\ y\}$;*
- *the squaring function: $t(R) = \{(x, z) \mid \exists y, x\ R\ y\ R\ z\}$.*

Intuitively, given a relation R, $(s \cup \mathrm{id})(R)$ is the symmetric closure of R, $(r \cup s \cup \mathrm{id})(R)$ is its reflexive and symmetric closure, and $(r \cup s \cup t \cup \mathrm{id})^\omega(R)$ is its symmetric, reflexive and transitive closure: $e = (r \cup s \cup t \cup \mathrm{id})^\omega$. Another way to understand this decomposition of e is to recall that for a given R, $e(R)$ can be defined inductively by the following rules:

$$\frac{}{x\ e(R)\ x}\ r \qquad \frac{x\ e(R)\ y}{y\ e(R)\ x}\ s \qquad \frac{x\ e(R)\ y \quad y\ e(R)\ z}{x\ e(R)\ z}\ t \qquad \frac{x\ R\ y}{x\ e(R)\ y}\ \mathrm{id}$$

Theorem 1. *Any bisimulation up to e is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that e is compatible, which follows from Lemma 1 and Lemma 2. □

Corollary 1. *For all $x, y \in S$, $x \sim y$ iff* $\mathtt{HK}(x, y)$.

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail e(R) \cup todo$. We deduce that R is a bisimulation up to e after the loop. We conclude with Theorem 1 and Proposition 1. □

Returning to the right-hand side example from Figure 2, Hopcroft and Karp's algorithm constructs the relation

$$R_{\mathtt{HK}} = \{(x, u), (y, v), (z, w), (z, v)\}$$

which is not a bisimulation, but a bisimulation up to e: it contains the pair (x, u), whose b-transitions lead to (y, w), which is not in $R_{\mathtt{HK}}$ but in its equivalence closure, $e(R_{\mathtt{HK}})$.

3. Optimised algorithm for NFA

We now move from DFA to non-deterministic automata (NFA). We start with standard definitions about semi-lattices, determinisation, and language equivalence for NFA.

A *semi-lattice* $(X, +, 0)$ consists of a set X and a binary operation $+ \colon X \times X \to X$ which is associative, commutative, idempotent (ACI), and has $0 \in X$ as identity. Given two semi-lattices $(X_1, +_1, 0_1)$ and $(X_2, +_2, 0_2)$, an *homomorphism* of semi-lattices is a function $f \colon X_1 \to X_2$ such that for all $x, y \in X_1$, $f(x +_1 y) = f(x) +_2 f(y)$ and $f(0_1) = 0_2$. The set $2 = \{0, 1\}$ is a semi-lattice when taking $+$ to be the ordinary Boolean or. Also the set of all languages $2^{A^\star}$ carries a semi-lattice where $+$ is the union of languages and 0 is the empty language. More generally, for any set X, $\mathcal{P}(X)$ is a semi-lattice where $+$ is the union of sets and 0 is the empty set. In the sequel, we indiscriminately use 0 to denote the element $0 \in 2$, the empty language in $2^{A^\star}$, and the empty set in $\mathcal{P}(X)$. Similarly, we use $+$ to denote the Boolean or in 2, the union of languages in $2^{A^\star}$, and the union of sets in $\mathcal{P}(X)$.

A non-deterministic finite automaton (NFA) over the input alphabet A is a triple (S, o, t), where S is a finite set of states, $o \colon S \to 2$ is the output function (as for DFA), and $t \colon S \to \mathcal{P}(S)^A$ is the transition relation, which assigns to each state $x \in S$ and input letter $a \in A$ a set of possible successor states.

The *powerset construction* transforms any NFA (S, o, t) in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ where $o^\sharp \colon \mathcal{P}(S) \to 2$ and $t^\sharp \colon \mathcal{P}(S) \to \mathcal{P}(S)^A$ are defined for all $X \in \mathcal{P}(S)$ and $a \in A$ as follows:

$$o^\sharp(X) = \begin{cases} o(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ o^\sharp(X_1) + o^\sharp(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

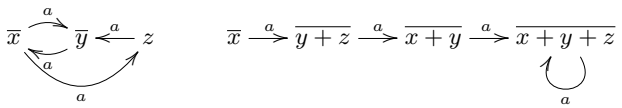
$$t^\sharp_a(X) = \begin{cases} t_a(x) & \text{if } X = \{x\} \text{ with } x \in S \\ 0 & \text{if } X = 0 \\ t^\sharp_a(X_1) + t^\sharp_a(X_2) & \text{if } X = X_1 + X_2 \end{cases}$$

Observe that in $(\mathcal{P}(S), o^\sharp, t^\sharp)$, the states form a semi-lattice $(\mathcal{P}(S), +, 0)$, and $o^\sharp$ and $t^\sharp$ are, by definition, semi-lattices homomorphisms. These properties are fundamental for the up-to technique we are going to introduce; in order to highlight the difference with generic DFA (which usually do not carry this structure), we introduce the following definition.

Definition 4. *A* determinised NFA *is a DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ obtained via the powerset construction of some NFA (S, o, t).*

Hereafter, we use a new notation for representing states of determinised NFA: in place of the singleton $\{x\}$ we just write x and, in place of $\{x_1, \ldots, x_n\}$, we write $x_1 + \cdots + x_n$.

For an example, consider the NFA (S, o, t) depicted below (left) and part of the determinised NFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$ (right).



In the determinised NFA, x makes one single a-transition going into $y + z$. This state is final: $o^\sharp(y + z) = o^\sharp(y) + o^\sharp(z) = o(y) + o(z) = 1 + 0 = 1$; it makes an a-transition into $t^\sharp_a(y + z) = t^\sharp_a(y) + t^\sharp_a(z) = t_a(y) + t_a(z) = x + y$.

The language accepted by the states of a NFA (S, o, t) can be conveniently defined via the powerset construction: the language accepted by $x \in S$ is the language accepted by the singleton $\{x\}$ in the DFA $(\mathcal{P}(S), o^\sharp, t^\sharp)$, in symbols $[\![\{x\}]\!]$. Therefore, in the following, instead of considering the problem of language equivalence of states of the NFA, we focus on language equivalence of *sets* of states of the NFA: given two sets of states X and Y in $\mathcal{P}(S)$, we say that X and Y are language equivalent ($X \sim Y$) iff $[\![X]\!] = [\![Y]\!]$. This is exactly what happens in standard automata theory, where NFA are equipped with sets of initial states.

3.1 Extending coinduction to NFA

In order to check if two sets of states X and Y of an NFA (S, o, t) are language equivalent, we can simply employ the bisimulation proof method on $(\mathcal{P}(S), o^\sharp, t^\sharp)$. More explicitly, a bisimulation for a NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then (1) $o^\sharp(X) = o^\sharp(Y)$, and (2) for all $a \in A$, $t^\sharp_a(X)\ R\ t^\sharp_a(Y)$. Since this is just the old definition of bisimulation (Definition 1) applied to $(\mathcal{P}(S), o^\sharp, t^\sharp)$, we get that $X \sim Y$ iff there exists a bisimulation relating them.

hal-00639716, version 5 - 11 Jul 2012

Remark 3 (Linear time v.s. branching time)**.** *It is important not to confuse these bisimulation relations with the standard Milner-and-Park bisimulations [24] (which* strictly *imply language equivalence): in a standard bisimulation R, if the following states x and y of an NFA are in R,*

then each x_i should be in R with some y_j (and vice-versa). Here, instead, we first transform the transition relation into

$$x \xrightarrow{a} x_1 + \cdots + x_n \qquad y \xrightarrow{a} y_1 + \cdots + y_m \ ,$$

using the powerset construction, and then we require that the sets $x_1 + \cdots + x_n$ and $y_1 + \cdots + y_m$ are related by R.

3.2 Bisimulation up to congruence

The semi-lattice structure $(\mathcal{P}(S), +, 0)$ carried by determinised NFA makes it possible to introduce a new up-to technique, which is not available with plain DFA: *up to congruence*. This technique relies on the following function.

Definition 5 (Congruence closure)**.** *Let $u\colon \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S))$ be the function on relations on sets of states defined for all $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as:*

$$u(R) = \{(X_1 + X_2,\ Y_1 + Y_2) \mid X_1\ R\ Y_1 \text{ and } X_2\ R\ Y_2\}\,.$$

The function $c = (r \cup s \cup t \cup u \cup \mathrm{id})^\omega$ is called the congruence closure *function.*

Intuitively, $c(R)$ is the smallest equivalence relation which is closed with respect to $+$ and which includes R. It could alternatively be defined inductively using the rules r, s, t, and id from the previous section, and the following one:

$$\frac{X_1\ c(R)\ Y_1 \qquad X_2\ c(R)\ Y_2}{X_1 + X_2\ c(R)\ Y_1 + Y_2}\ u$$

We call bisimulations up to congruence the bisimulations up to c. We report the explicit definition for the sake of clarity:

Definition 6 (Bisimulation up to congruence)**.** *A bisimulation up to congruence for a NFA (S, o, t) is a relation $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, such that whenever $X\ R\ Y$ then*

1. *$o^\sharp(X) = o^\sharp(Y)$ and*
2. *for all $a \in A$, $t_a^\sharp(X)\ c(R)\ t_a^\sharp(Y)$.*

We then show that bisimulations up to congruence are sound, using the notion of compatibility:

Lemma 3. *The function u is compatible.*

Proof. We assume that $R \rightarrowtail R'$, and we prove that $u(R) \rightarrowtail u(R')$. If $X\ u(R)\ Y$, then $X = X_1 + X_2$ and $Y = Y_1 + Y_2$ for some X_1, X_2, Y_1, Y_2 such that $X_1\ R\ Y_1$ and $X_2\ R\ Y_2$. By assumption, we have $o^\sharp(X_1) = o^\sharp(Y_1)$, $o^\sharp(X_2) = o^\sharp(Y_2)$, and for all $a \in A$, $t_a^\sharp(X_1)\ R'\ t_a^\sharp(Y_1)$ and $t_a^\sharp(X_2)\ R'\ t_a^\sharp(Y_2)$. Since $o^\sharp$ and $t^\sharp$ are homomorphisms, we deduce $o^\sharp(X_1 + X_2) = o^\sharp(Y_1 + Y_2)$, and for all $a \in A$, $t_a^\sharp(X_1 + X_2)\ u(R')\ t_a^\sharp(Y_1 + Y_2)$. □

Theorem 2. *Any bisimulation up to congruence is contained in a bisimulation.*

Proof. By Proposition 3, it suffices to show that c is compatible, which follows from Lemmas 1, 2 and 3. □

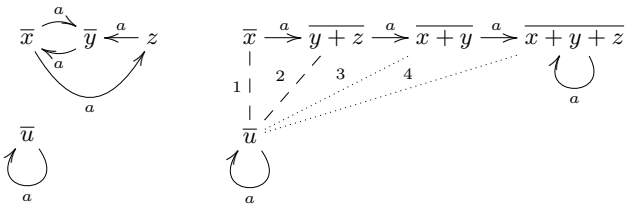


Figure 3. Bisimulations up to congruence, on a single letter NFA.

In the Introduction, we already gave an example of bisimulation up to context, which is a particular case of bisimulation up to congruence (up to context corresponds to use just the function $(r \cup u \cup \mathrm{id})^\omega$, without closing under s and t).

A more involved example illustrating the use of all ingredients of the congruence closure function (c) is given in Figure 3. The relation R expressed by the dashed numbered lines (formally $R = \{(x,\ u), (y + z,\ u)\}$) is neither a bisimulation, nor a bisimulation up to equivalence, since $y + z \xrightarrow{a} x + y$ and $u \xrightarrow{a} u$, but $(x+y,\ u) \notin e(R)$. However, R is a bisimulation up to congruence. Indeed, we have $(x + y,\ u) \in c(R)$:

$$\begin{aligned} x + y\ c(R)\ & u + y && ((x, u) \in R)\\ c(R)\ & y + z + y && ((y + z, u) \in R)\\ =\ & y + z\\ c(R)\ & u && ((y + z, u) \in R) \end{aligned}$$

In contrast, we need four pairs to get a bisimulation up to e containing (x, u): this is the relation depicted with both dashed and dotted lines in Figure 3.

Note that we can deduce many other equations from R; in fact, $c(R)$ defines the following partition of sets of states:

$$\{0\}, \{y\}, \{z\}, \{x, u, x+y, x+z, \text{ and the 9 remaining subsets}\}.$$

3.3 Optimised algorithm for NFA

Algorithms for NFA can be obtained by computing the determinised NFA on-the-fly [12]: starting from the algorithms for DFA (Figure 1), it suffices to work with sets of states, and to inline the powerset construction. The corresponding code is given in Figure 4. The naive algorithm (`Naive`) does not use any up to technique, Hopcroft and Karp's algorithm (`HK`) reasons up to equivalence in step `3.2`, and the optimised algorithm, referred as `HKC` in the sequel, relies on up to congruence: step `3.2` becomes

```
(3.2)  if (X′,Y′) ∈ c(R ∪ todo) then skip;
```

Observe that we use $c(R \cup todo)$ rather than $c(R)$: this allows us to skip more pairs, and this is safe since all pairs in $todo$ will eventually be processed.

Corollary 2. *For all $X, Y \in \mathcal{P}(S)$, $X \sim Y$ iff* `HKC`(X, Y).

Proof. Same proof as for Proposition 2, by using the invariant $R \rightarrowtail c(R \cup todo)$ for the loop. We deduce that R is a bisimulation up to congruence after the loop. We conclude with Theorem 2 and Proposition 1. □

The most important point about these three algorithms is that they compute the states of the determinised NFA lazily. This means that only *accessible* states need to be computed, which is of practical importance since the determinised NFA can be exponentially large. In case of a negative answer, the three algorithms stop even before all accessible states have been explored; otherwise, if a bisimulation (possibly up-to) is found, it depends on the algorithm:

hal-00639716, version 5 - 11 Jul 2012

$\underline{\texttt{Naive}(X, Y)}$

```
(1) R is empty; todo is empty;
(2) insert (X,Y) in todo;
(3) while todo is not empty, do {
 (3.1)  extract (X',Y') from todo;
 (3.2)  if (X',Y') ∈ R then skip;
 (3.3)  if o♯(X') ≠ o♯(Y') then return false;
 (3.4)  for all a ∈ A,
          insert (t_a♯(X'), t_a♯(Y')) in todo;
 (3.5)  insert (X',Y') in R;
(4) return true;
```

Figure 4. On-the-fly naive algorithm, for checking the equivalence of sets of states X and Y of a NFA (S, o, t). The code for on-the-fly $\texttt{HK}(X, Y)$ is obtained by replacing the test in step `3.2` with $(X', Y') \in e(R)$; the code for $\texttt{HKC}(X, Y)$ is obtained by replacing this test with $(X', Y') \in c(R \cup todo)$.

- with `Naive`, all accessible states need to be visited, by definition of bisimulation;
- with `HK`, the only case where some accessible states can be avoided is when a pair (X, X) is encountered: the algorithm skips this pair so that the successors of X are not necessarily computed (this situation rarely happens in practice—it actually never happens when starting with disjoint automata). In the other cases where a pair (X, Y) is skipped, then X and Y are necessarily already related to some other states in R, so that their successors will eventually be explored;
- with `HKC`, only a small portion of the accessible states is built (check the experiments in Section 6). To see a concrete example, let us execute `HKC` on the NFA from Figure 3. After two iterations, $R = \{(x, u), (y + z, u)\}$. Since $x + y\ c(R)\ u$, the algorithm stops without building the states $x + y$ and $x + y + z$. Similarly, in the example from the Introduction, `HKC` does not construct the four states corresponding to pairs 4, 5, and 6.

This ability of `HKC` to ignore parts of the determinised NFA comes from the up to congruence technique, which allows one to infer properties about states that were not necessarily encountered before. As we shall see in Section 4 the efficiency of antichains algorithms [1, 33] also comes from their ability to skip large parts of the determinised NFA.

3.4 Computing the congruence closure

For the optimised algorithm to be effective, we need a way to check whether some pairs belong to the congruence closure of some relation (step `3.2`). We present here a simple solution based on set rewriting; the key idea is to look at each pair (X, Y) in a relation R as a pair of rewriting rules:

$$X \to X + Y \qquad\qquad Y \to X + Y \ ,$$

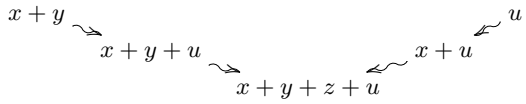
which can be used to compute normal forms for sets of states. Indeed, by idempotence, $X\ R\ Y$ entails $X\ c(R)\ X + Y$.

Definition 7. *Let $R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ be a relation on sets of states. We define $\rightsquigarrow_R \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ as the smallest irreflexive relation that satisfies the following rules:*

$$\frac{X\ R\ Y}{X \rightsquigarrow_R X + Y} \qquad \frac{X\ R\ Y}{Y \rightsquigarrow_R X + Y} \qquad \frac{Z \rightsquigarrow_R Z'}{U + Z \rightsquigarrow_R U + Z'}$$

Lemma 4. *For all relations R, the relation $\rightsquigarrow_R$ is convergent.*

In the sequel, we denote by $X{\downarrow_R}$ the normal form of a set X w.r.t. $\rightsquigarrow_R$. Intuitively, the normal form of a set is the largest set of its equivalence class. Recalling the example from Figure 3, the common normal form of $x + y$ and u can be computed as follows (R is the relation $\{(x, u), (y + z, u)\}$):



Theorem 3. *For all relations R, and for all $X, Y \in \mathcal{P}(S)$, we have $X{\downarrow_R} = Y{\downarrow_R}$ iff $(X, Y) \in c(R)$.*

Thus, in order to check if $(X, Y) \in c(R \cup todo)$ we only have to compute the normal form of X and Y with respect to $\rightsquigarrow_{R \cup todo}$. Note that each pair of $R \cup todo$ may be used only once as a rewriting rule, but we do not know in advance in which order to apply these rules. Therefore, the time required to find one rule that applies is in the worst case rn where $r = |R \cup todo|$ is the size of the relation $R \cup todo$, and $n = |S|$ is the number of states of the NFA (assuming linear time complexity for set-theoretic union and containment of sets of states). Since we cannot apply more than r rules, the time for checking whether $(X, Y) \in c(R \cup todo)$ is bounded by $r^2 n$.

We tried other solutions, notably by using binary decision diagrams [8]. We have chosen to keep the presented rewriting algorithm for its simplicity and because it behaves well in practice.

3.5 Complexity hints

The complexity of `Naive`, `HK` and `HKC` is closely related to the size of the relation that they build. Hereafter, we use $v = |A|$ to denote the number of letters in A.

Lemma 5. *The three algorithms require at most $1 + v{\cdot}|R|$ iterations, where $|R|$ is the size of the produced relation; moreover, this bound is reached whenever they return true.*

Therefore, we can conveniently reason about $|R|$.

Lemma 6. *Let $R_{\texttt{Naive}}$, $R_{\texttt{HK}}$, and $R_{\texttt{HKC}}$ denote the relations produced by the three algorithms. We have*

$$|R_{\texttt{HKC}}|, |R_{\texttt{HK}}| \leq m \qquad\qquad |R_{\texttt{Naive}}| \leq m^2 \ , \tag{2}$$

where $m \leq 2^n$ is the number of accessible states in the determinised NFA and n is the number of states of the NFA. If the algorithms returned true, we moreover have

$$|R_{\texttt{HKC}}| \leq |R_{\texttt{HK}}| \leq |R_{\texttt{Naive}}| \ . \tag{3}$$

As shown below in Section 4.2.4, $R_{\texttt{HKC}}$ can be exponentially smaller than $R_{\texttt{HK}}$. Notice however that the problem of deciding NFA language equivalence is PSPACE-complete [23], and that none of the algorithms presented here is in PSPACE: all of them store a set of visited pairs, and in the worst case, this set can become exponentially large with all of them. (This also holds for the antichain algorithms [1, 33] which we describe in Section 4.) Instead, the standard PSPACE algorithm does not store any set of visited pairs: it checks all words of length smaller than 2^n. While this can be done in polynomial space, this systematically requires exponential time.

3.6 Using `HKC` for checking language inclusion

For NFA, language inclusion can be reduced to language equivalence in a rather simple way. Since the function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^\star}$ is a semi-lattice homomorphism (see Theorem 7 in Appendix A), for any given sets of states X and Y, $[\![X + Y]\!] = [\![Y]\!]$ iff $[\![X]\!] + [\![Y]\!] = [\![Y]\!]$ iff $[\![X]\!] \subseteq [\![Y]\!]$. Therefore, it suffices to run $\texttt{HKC}(X + Y, Y)$ to check the inclusion $[\![X]\!] \subseteq [\![Y]\!]$.

In such a situation, all pairs that are eventually manipulated by `HKC` have the shape $(X' + Y', Y')$ for some sets X', Y'. The step `3.2` of `HKC`, where it checks whether the current pair belongs

hal-00639716, version 5 - 11 Jul 2012

to the congruence closure of the relation, can thus be simplified. First, the pairs in the current relation can only be used to rewrite from right to left. Second, the following lemma allows one to avoid unnecessary normal form computations:

Lemma 7. *For all sets X, Y and for all relations R, we have $X+Y\ c(R)\ Y$ iff $X \subseteq Y{\downarrow_R}$.*

Proof. We first prove that for all X, Y, $X{\downarrow_R} = Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$ and $Y \subseteq X{\downarrow_R}$, using the fact that the normalisation function ${\downarrow_R}\colon\ X \mapsto X{\downarrow_R}$ is monotone and idempotent. The announced result follows by Theorem 3, since $Y \subseteq (X+Y){\downarrow_R}$ is always true and $X+Y \subseteq Y{\downarrow_R}$ iff $X \subseteq Y{\downarrow_R}$. □

However, as shown below, checking an equivalence by decomposing it into two inclusions cannot be more efficient than checking the equivalence directly.

Lemma 8. *Let X, Y be two sets of states; let $R_\subseteq$ and $R_\supseteq$ be the relations computed by HKC$(X+Y, Y)$ and HKC$(X+Y, X)$, respectively. If $R_\subseteq$ and $R_\supseteq$ are bisimulations up to congruence, then the following relation is a bisimulation up to congruence:*

$$R_= = \{(X', Y') \mid (X'+Y', Y') \in R_\subseteq \text{ or } (X'+Y', X') \in R_\supseteq\}.$$

On the contrary, checking the equivalence directly actually allows one to skip some pairs that cannot be skipped when reasoning by double inclusion. As an example, consider the DFA on the right of Figure 2. The relation computed by HKC(x, u) contains only four pairs (because the fifth one follows from transitivity). Instead, the relations built by HKC$(x, x+u)$ and HKC$(u+x, u)$ would both contain five pairs: transitivity cannot be used since our relations are now oriented (from $y \leq v$, $z \leq v$ and $z \leq w$, we cannot deduce $y \leq w$). Another example, where we get an exponential factor by checking the equivalence directly rather than through the two inclusions, can be found in Section 4.2.4.

In a sense, the behaviour of the coinduction proof method here is similar to that of standard proofs by induction, where one often has to strengthen the induction predicate to get a (nicer) proof.

4. Antichain algorithm

In [33], De Wulf et al. have proposed the *antichain* approach for checking language inclusion of NFA. We show that this approach can be explained in terms of *simulations up to upward-closure* that, in turn, can be seen as a special case of bisimulations up to congruence. Before doing so, we recall the standard notion of antichain and we describe the antichain algorithm (AC).

Given a partial order $(X, \sqsubseteq)$, an *antichain* is a subset $Y \subseteq X$ containing only incomparable elements (that is, for all $y_1, y_2 \in Y$, $y_1 \not\sqsubseteq y_2$ and $y_2 \not\sqsubseteq y_1$). AC exploits antichains over the set $S \times \mathcal{P}(S)$, where the ordering is given by $(x_1, Y_1) \sqsubseteq (x_2, Y_2)$ iff $x_1 = x_2$ and $Y_1 \subseteq Y_2$.

In order to check $[\![X]\!] \subseteq [\![Y]\!]$ for two sets of states X, Y of an NFA (S, o, t), AC maintains an antichain of pairs (x', Y'), where x' is a state of the NFA and Y' is a state of the determinised automaton. More precisely, the automaton is explored non-deterministically (via t) for obtaining the first component of the pair and deterministically (via $t^\sharp$) for the second one. If a pair such that x' is accepting $(o(x') = 1)$ and Y' is not $(o^\sharp(Y') = 0)$ is encountered, then a counter-example has been found. Otherwise all derivatives of the pair along the automata transitions have to be inserted into the antichain, so that they will be explored. If one these pairs p is larger than a previously encountered pair p' $(p' \sqsubseteq p)$ then the language inclusion corresponding to p is subsumed by p' so that p can be skipped; otherwise, if $p \sqsubseteq p_1, \ldots, p_n$ for some pairs $p_1, \ldots, p_n$ that are already in the antichain, then one can safely remove these pairs: they are subsumed by p and, by doing so, the set of visited pairs remains an antichain.

Remark 4. *An important difference between HKC and AC consists in the fact that the former inserts pairs in todo without checking whether they are redundant (this check is performed when the pair is processed), while the latter removes all redundant pairs whenever a new one is inserted. Therefore, the cost of an iteration with HKC is merely the cost of the corresponding congruence check, while the cost of an iteration with AC is merely that of inserting all successors of the corresponding pair and simplifying the antichain.*

Note that the above description corresponds to the "forward" antichain algorithm, as described in [1]. Instead, the original antichain algorithm, as first described in [33], is "backward" in the sense that the automata are traversed in the reversed way, from accepting states to initial states. The two versions are dual [33] and we could similarly define the backward counterpart of HKC and HK. We however stick to the forward versions for the sake of clarity.

4.1 Coinductive presentation

Leaving apart the concrete data structures used to manipulate antichains, we can rephrase this algorithm using a coinductive framework, like we did for Hopcroft and Karp's algorithm.

First define a notion of *simulation*, where the left-hand side automaton is executed non-deterministically:

Definition 8 (Simulation)**.** *Given two relations $T, T' \subseteq S \times \mathcal{P}(S)$, T s-progresses to T', denoted $T \rightarrowtail_s T'$, if whenever $x\ T\ Y$ then*

1. *$o(x) \leq o^\sharp(Y)$ and*
2. *for all $a \in A$, $x'\ T'\ t_a^\sharp(Y)$.*

A simulation *is a relation T such that $T \rightarrowtail_s T$.*

As expected, we obtain the following coinductive proof principle:

Proposition 4 (Coinduction)**.** *For all sets X, Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff there exists a simulation T such that for all $x \in X$, $x\ T\ Y$.*

(Note that like for our notion of bisimulation, the above notion of simulation is weaker than the standard one from concurrency theory [24], which *strictly* entails language inclusion—Remark 3.)

To account for the antichain algorithm, where we can discard pairs using the preorder $\sqsubseteq$, it suffices to define the *upward closure* function $\uparrow\colon \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ as

$$\uparrow T = \{(x, Y) \mid \exists (x', Y') \in T \text{ s.t. } (x', Y') \sqsubseteq (x, Y)\}\ .$$

A pair belongs to the upward closure $\uparrow T$ of a relation $T \subseteq S \times \mathcal{P}(S)$, if and only if this pair is subsumed by some pair in T. In fact, rather than trying to construct a simulation, AC attempts to construct a simulation up to upward closure.

Like for HK and HKC, this method can be justified by defining the appropriate notion of s-compatible function, showing that any simulation up to an s-compatible function is contained in a simulation, and showing that the upward closure function $(\uparrow)$ is s-compatible.

Theorem 4. *Any simulation up to $\uparrow$ is contained in a simulation.*

Corollary 3. *For all $X, Y \in \mathcal{P}(S)$, $[\![X]\!] \subseteq [\![Y]\!]$ iff* AC(X, Y).

4.2 Comparing HKC and AC

The efficiency of the two algorithms strongly depends on the number of pairs that they need to explore. In the following (Sections 4.2.3 and 4.2.4), we show that HKC can explore far fewer pairs than AC, when checking language inclusion of automata that share some states, or when checking language equivalence. We would also like to formally prove that (a) HKC never explores more than AC, and

hal-00639716, version 5 - 11 Jul 2012

(b) when checking inclusion of disjoint automata, AC never explores more than HKC. Unfortunately, the validity of these statements highly depends on numerous assumptions about the two algorithms (e.g., on the exploration strategy) and their potential proofs seem complicated and not really informative. For these reasons, we preferred to investigate the formal correspondence at the level of the coinductive proof techniques, where it is much cleaner.

4.2.1 Language inclusion: HKC can mimic AC

As explained in Section 3.6, we can check the language inclusion of two sets X, Y by executing $\mathtt{HKC}(X{+}Y, Y)$. We now show that for any simulation up to upward closure that proves the inclusion $[\![X]\!] \subseteq [\![Y]\!]$, there exists a bisimulation up to congruence of the same size which proves the same inclusion. For $T \subseteq S \times \mathcal{P}(S)$, let $\widehat{T} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ denote the relation $\{(x+Y, Y) \mid x\ T\ Y\}$.

Lemma 9. *We have* $\widehat{\uparrow T} \subseteq c(\widehat{T})$.

Proof. If $(x+Y, Y) \in \widehat{\uparrow T}$, then there exists $Y' \subseteq Y$ such that $(x, Y') \in T$. By definition, $(x+Y', Y') \in \widehat{T}$ and $(Y, Y) \in c(\widehat{T})$. By the rule (u), $(x+Y'+Y, Y'+Y) \in c(\widehat{T})$ and since $Y' \subseteq Y$, $(x+Y, Y) \in c(\widehat{T})$. □

Proposition 5. *If T is a simulation up to $\uparrow$, then $\widehat{T}$ is a bisimulation up to c.*

Proof. First observe that if $T \rightarrowtail_s T'$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{T'})$. Therefore, if $T \rightarrowtail_s \uparrow T$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{\uparrow T})$. By Lemma 9, $\widehat{T} \rightarrowtail u^\omega(c(\widehat{T})) = c(\widehat{T})$. □

(Note that transitivity and symmetry are not used in the above proofs: the constructed bisimulation up to congruence is actually a bisimulation up to context $(r \cup u \cup id)^\omega$.)

The relation $\widehat{T}$ is not the one computed by HKC, since the former contains pairs of the shape $(x+Y, Y)$, while the latter has pairs of the shape $(X+Y, Y)$ with X possibly not a singleton. However, note that manipulating pairs of the two kinds does not change anything since by Lemma 7, $(X+Y, Y) \in c(R)$ iff for all $x \in X$, $(x+Y, Y) \in c(R)$.

4.2.2 Inclusion: AC can mimic HKC on disjoint automata

As shown in Section 4.2.3 below, HKC can be faster than AC, thanks to the up to transitivity technique. However, in the special case where the two automata are disjoint, transitivity cannot help, and the two algorithms actually match each other.

Suppose that the automaton (S, o, t) is built from two disjoint automata (S_1, o_1, t_1) and (S_2, o_2, t_2) as described in Remark 1. Let R be the relation obtained by running $\mathtt{HKC}(X_0{+}Y_0, Y_0)$ with $X_0 \subseteq S_1$ and $Y_0 \subseteq S_2$. All pairs in R are necessarily of the shape $(X{+}Y,\ Y)$ with $X \subseteq S_1$ and $Y \subseteq S_2$. Let $\overline{R} \subseteq S \times \mathcal{P}(S)$ denote the relation $\{(x, Y) \mid \exists X,\ x \in X \text{ and } X{+}Y\ R\ Y\}$.

Lemma 10. *If S_1 and S_2 are disjoint, then $\overline{c(R)} \subseteq \uparrow(\overline{R})$.*

Proof. Suppose that $x\ \overline{c(R)}\ Y$, i.e., $x \in X$ with $X + Y\ c(R)\ Y$. By Lemma 7, we have $X \subseteq Y{\downarrow_R}$, and hence, $x \in Y{\downarrow_R}$. By definition of R the pairs it contains can only be used to rewrite from right to left; moreover, since S_1 and S_2 are disjoint, such rewriting steps cannot enable new rewriting rules, so that all steps can be performed in parallel: we have $Y{\downarrow_R} = \sum_{X'+Y'\ R\ Y' \subseteq Y} X'$. Therefore, there exists some X', Y' with $x \in X'$, $X'{+}Y'\ R\ Y'$, and $Y' \subseteq Y$. It follows that $(x, Y') \in \overline{R}$, hence $(x, Y) \in \uparrow(\overline{R})$. □

Proposition 6. *If S_1 and S_2 are disjoint, and if R is a bisimulation up to congruence, then $\overline{R}$ is a simulation up to upward closure.*

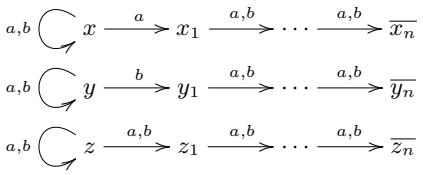


Figure 5. Family of examples where HKC exponentially improves over AC and HK; we have $x + y \sim z$.

Proof. First observe that for all relations R, R', if $R \rightarrowtail R'$, then $\overline{R} \rightarrowtail_s \overline{R'}$. Therefore, if $R \rightarrowtail c(R)$, then $\overline{R} \rightarrowtail_s \overline{c(R)}$. We deduce $\overline{R} \rightarrowtail_s \uparrow(\overline{R})$ by Lemma 10. □

4.2.3 Inclusion: AC cannot mimic HKC on merged automata

The containment of Lemma 10 does not hold when S_1 and S_2 are not disjoint, since c can exploit transitivity, while $\uparrow$ cannot. For a concrete grasp, take $R = \{(x+y, y), (y+z, z)\}$ and observe that $(x, z) \in \overline{c(R)}$ but $(x, z) \notin \uparrow(\overline{R})$. This difference makes it possible to find bisimulations up to c that are much smaller than the corresponding simulations up to $\uparrow$, and for HKC to be more efficient than AC. Such an example, where HKC is exponentially better than AC for checking language inclusion of automata sharing some states, is given in [6].

4.2.4 Language equivalence: AC cannot mimic HKC.

AC can be used to check language equivalence, by checking the two underlying inclusions. However, checking equivalence directly can be better, even in the disjoint case. To see this on a simple example, consider the DFA on the right-hand side of Figure 2. If we use AC twice to prove $x \sim u$, we get the following antichains

$$T_1 = \{(x, u), (y, v), (y, w), (z, v), (z, w)\}\ ,$$
$$T_2 = \{(u, x), (v, y), (w, y), (v, z), (w, z)\}\ ,$$

containing five pairs each. Instead, four pairs are sufficient with HK or HKC, thanks to up to symmetry and up to transitivity.

For a more interesting example, consider the family of NFA given in Figure 5, where n is an arbitrary natural number. Taken together, the states x and y are equivalent to the state z: they recognise the language $(a{+}b)^\star(a{+}b)^{n+1}$. Alone, the state x (resp. y) recognises the language $(a{+}b)^\star a(a{+}b)^n$ (resp. $(a{+}b)^\star b(a{+}b)^n$).

For $i \leq n$, let $X_i = x{+}x_1{+}\ldots{+}x_i$, $Y_i = y{+}y_1{+}\ldots{+}y_i$, and $Z_i = z{+}z_1{+}\ldots{+}z_i$; for $N \subseteq [1..i]$, furthermore set

$$X_i^N = x + \sum_{j \in N} x_j\ , \qquad \overline{Y}_i^N = y + \sum_{j \in [1..n] \setminus N} y_j\ .$$

In the determinised NFA, $x + y$ can reach all the states of the shape $X_i^N{+}\overline{Y}_i^N$, for $i \leq n$ and $N \subseteq [1..i]$. For instance, for $n{=}i{=}2$, we have $x{+}y \stackrel{aa}{\rightarrow} x{+}y{+}x_1{+}x_2$, $x{+}y \stackrel{ab}{\rightarrow} x{+}y{+}y_1{+}x_2$, $x{+}y \stackrel{ba}{\rightarrow} x{+}y{+}x_1{+}y_2$, and $x{+}y \stackrel{bb}{\rightarrow} x{+}y{+}y_1{+}y_2$. Instead, z reaches only $n{+}1$ distinct states, those of the form Z_i.

The smallest bisimulation relating $x + y$ and z is

$$R = \{(X_i^N + \overline{Y}_i^N,\ Z_i) \mid i \leq n, N \subseteq [1..i]\},$$

which contains $2^{n+1}{-}1$ pairs. This is the relation computed by $\mathtt{Naive}(x, y)$ and $\mathtt{HK}(x, y)$—the up to equivalence technique (alone) does not help in HK. With AC, we obtain the antichains $T_x + T_y$ (for

hal-00639716, version 5 - 11 Jul 2012

$[\![x+y]\!] \subseteq [\![z]\!]$) and T_z (for $[\![x+y]\!] \supseteq [\![z]\!]$), where:

$$T_x = \{(x_i,\ Z_i) \mid i \leq n\},$$
$$T_y = \{(y_i,\ Z_i) \mid i \leq n\},$$
$$T_z = \{(z_i,\ X_i^N + \overline{Y}_i^N) \mid i \leq n, N \subseteq [1..i]\}.$$

Note that T_x and T_y have size $n+1$, and T_z has size $2^{n+1}-1$.

The language recognised by x or y are known for having a minimal DFA with 2^n states [17]. So, checking $x+y \sim z$ via minimisation (e.g., [9, 15]) would also require exponential time.

This is not the case with HKC, which requires only polynomial time in this case. Indeed, HKC$(x+y, z)$ builds the relation

$$R' = \{(x+y,\ z)\}$$
$$\cup\, \{(x + Y_i + y_{i+1},\ Z_{i+1}) \mid i < n\}$$
$$\cup\, \{(x + Y_i + x_{i+1},\ Z_{i+1}) \mid i < n\}$$

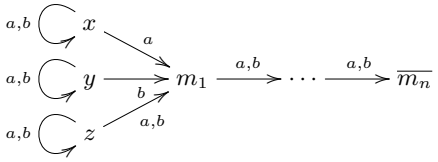
which is a bisimulation up to congruence and which only contains $2n+1$ pairs. To see that this is a bisimulation up to congruence, consider the pair $(x+y+x_1+y_2,\ Z_2)$ obtained from $(x+y,\ z)$ after reading the word ba. This pair does not belong to R' but to its congruence closure. Indeed, we have

$$\begin{array}{lll} x+y+x_1+y_2\ c(R')\ Z_1+y_2 & & (x+y+x_1\ R'\ Z_1) \\ \qquad\qquad c(R')\ x+y+y_1+y_2 & & (x+y+y_1\ R'\ Z_1) \\ \qquad\qquad c(R')\ Z_2 & & (x+y+y_1+y_2\ R'\ Z_2) \end{array}$$

(Check Lemma 18 in Appendix D for a complete proof.)

5. Exploiting Similarity

Looking at the example in Figure 5, a natural idea would be to first quotient the automaton by graph isomorphism. By doing so, we would merge the states x_i, y_i, z_i, and we would obtain the following automaton, for which checking $x+y \sim z$ is much easier.



As shown by Abdulla et al. [1], one can actually do better with the antichain algorithm, by exploiting any preorder contained in language inclusion (e.g., similarity [24]). In this section, we rephrase this technique for antichains in our coinductive framework, and we show how this idea can be embedded in HKC, resulting in an even stronger algorithm.

5.1 AC with similarity: AC'

For the sake of clarity, we fix the preorder to be *similarity*, which can be computed in quadratic time [13]:

Definition 9 (Similarity)**.** Similarity *is the largest relation on states $\preceq \subseteq S \times S$ such that $x \preceq y$ entails:*

1. *$o(x) \leq o(y)$ and*
2. *for all $a \in A, x' \in S$ such that $x \xrightarrow{a} x'$, there exists some y' such that $y \xrightarrow{a} y'$ and $x' \preceq y'$.*

One extends similarity to a preorder $\preceq^{\forall\exists} \subseteq \mathcal{P}(S) \times \mathcal{P}(S)$ on sets of states, and to a preorder $\sqsubseteq^{\preceq} \subseteq (S \times \mathcal{P}(S)) \times (S \times \mathcal{P}(S))$ on antichain pairs, as:

$$X \preceq^{\forall\exists} Y \quad \text{if} \quad \forall x \in X, \exists y \in Y, x \preceq y\ ,$$
$$(x', Y') \sqsubseteq^{\preceq} (x, Y) \quad \text{if} \quad x \preceq x' \text{ and } Y' \preceq^{\forall\exists} Y\ .$$

The new antichain algorithm [1], which we call AC', is similar to AC, but the antichain is now taken w.r.t. the new preorder $\sqsubseteq^{\preceq}$. Formally, let $\nwarrow\!\!\!\uparrow : \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ be the function defined for all relations $T \subseteq S \times \mathcal{P}(S)$, as

$$\nwarrow\!\!\!\uparrow T = \{(x, Y) \mid x \preceq^{\forall\exists} Y, \text{ or}$$
$$\exists (x', Y') \in T \text{ s.t. } (x', Y') \sqsubseteq^{\preceq} (x, Y)\}.$$

While AC consists in trying to build a simulation up to $\uparrow$, AC' tries to build a simulation up to $\nwarrow\!\!\!\uparrow$, i.e., it skips a pair (x, Y) if either (a) it is subsumed by another pair of the antichain or (b) $x \preceq^{\forall\exists} Y$.

Theorem 5. *Any simulation up to $\nwarrow\!\!\!\uparrow$ is contained in a simulation.*

Corollary 4. *The antichain algorithm proposed in [1] is sound and complete: for all sets X, Y, $[\![X]\!] \subseteq [\![Y]\!]$ iff* AC'(X, Y).

Optimisation 1(a) and optimisation 1(b) in [1] are simply (a) and (b), as discussed above. Another optimisation, called Optimisation 2, is presented in [1]: if $y_1 \preceq y_2$ and $y_1, y_2 \in Y$ for some pair (x, Y), then y_1 can be safely removed from Y. Note that while this is useful to store smaller sets, it does not allow one to explore less, since the pairs encountered with or without optimisation 2 are always equivalent w.r.t. the ordering $\sqsubseteq^{\preceq}$: $Y \preceq^{\forall\exists} Y \setminus y_1$ and, for all $a \in A$, $t_a^\sharp(Y) \preceq^{\forall\exists} t_a^\sharp(Y \setminus y_1)$.

5.2 HKC with similarity: HKC'

Although HKC is primarily designed to check language equivalence, we can also extend it to exploit the similarity preorder. It suffices to notice that for any similarity pair $x \preceq y$, we have $x+y \sim y$.

Let $\overline{\preceq}$ denote the relation $\{(x+y,\ y) \mid x \preceq y\}$, let r' denote the constant to $\overline{\preceq}$ function, and let $c' = (r' \cup s \cup t \cup u \cup id)^\omega$. Accordingly, we call HKC' the algorithm obtained from HKC (Figure 4) by replacing $(X, Y) \in c(R \cup todo)$ with $(X, Y) \in c'(R \cup todo)$ in step 3.2. Notice that the latter test can be reduced to rewriting thanks to Theorem 3 and the following lemma.

Lemma 11. *For all relations R, $c'(R) = c(R \cup \overline{\preceq})$.*

In other words to check whether $(X, Y) \in c'(R \cup todo)$, it suffices to compute the normal forms of X and Y w.r.t. the rules from $R \cup todo$ plus the rules $x + y \leftarrow y$ for all $x \preceq y$.

Theorem 6. *Any bisimulation up to c' is contained in a bisimulation.*

Proof. Consider the constant function $r'' : \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S)) \to \mathcal{P}(\mathcal{P}(S) \times \mathcal{P}(S))$ mapping all relations to $\sim$. Since language equivalence $(\sim)$ is a bisimulation, we immediately obtain that this function is compatible. Thus so is the function $c'' = (r'' \cup s \cup t \cup u \cup id)^\omega$. We have that $\overline{\preceq}$ is contained in $\sim$, so that any bisimulation up to c' is a bisimulation up to c''. Since c'' is compatible, such a relation is contained in a bisimulation, by Proposition 3. □

Note that in the above proof, we can replace $\overline{\preceq}$ by any other relation contained in $\sim$. Intuitively, bisimulations up to c'' correspond to classical *bisimulations up to bisimilarity* [24] from concurrency.

Corollary 5. *For all sets X, Y, we have $X \sim Y$ iff* HKC'(X, Y).

5.3 Relationship between HKC' and AC'

Like in Section 4.2.1, we can show that for any simulation up to $\nwarrow\!\!\!\uparrow$ there exists a corresponding bisimulation up to c', of the same size.

Lemma 12. *For all relations $T \subseteq S \times \mathcal{P}(S)$, $\widehat{\nwarrow\!\!\!\uparrow T} \subseteq c'(\widehat{T})$.*

Proposition 7. *If T is a simulation up to $\nwarrow\!\!\!\uparrow$, then $\widehat{T}$ is a bisimulation up to c'.*

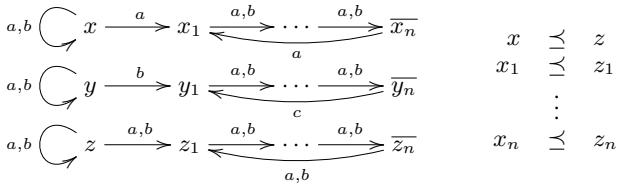


Figure 6. Family of examples where HKC' exponentially improves over AC', for inclusion of disjoint automata: we have $[\![z]\!] \subseteq [\![x+y]\!]$.

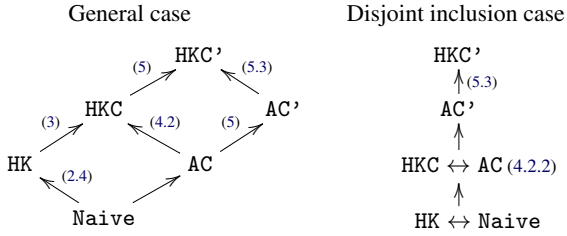


Figure 7. Relationship between the various algorithms.

However, even for checking inclusion of disjoint automata, AC' cannot mimic HKC', because now the similarity relation allows one to exploit transitivity. To see this, consider the example given in Figure 6, where we want to check that $[\![z]\!] \subseteq [\![x+y]\!]$, and for which the similarity relation is shown on the right-hand side.

Since this is an inclusion of disjoint automata, HKC and AC, which do not exploit similarity, behave the same (cf. Sections 4.2.1 and 4.2.2). Actually, they also behave like HK and they require $2^{n+1}-1$ pairs. On the contrary, the use of similarity allows HKC' to prove the inclusion with only $2n+1$ pairs, by computing the following bisimulation up to c' (Lemma 19 in Appendix E):

$$\begin{aligned} R'' = {} & \{(z{+}x{+}y,\ x{+}y)\} \\ & \cup \{(Z_{i+1}{+}X_i{+}y{+}y_{i+1},\ X_i{+}y{+}y_{i+1}) \mid i < n\} \\ & \cup \{(Z_{i+1}{+}X_{i+1}{+}y,\ X_{i+1}{+}y) \mid i < n\}\ , \end{aligned}$$

where $X_i = x{+}x_1{+}\ldots{+}x_i$ and $Z_i = z{+}z_1{+}\ldots{+}z_i$.

Like in Section 4.2.4, to see that this is a bisimulation up to c' (where we do exploit similarity), consider the pair obtained after reading the word ab: $(Z_2{+}x{+}y{+}x_2{+}y_1,\ x{+}y{+}x_2{+}y_1)$. This pair does not belong to R'' or $c(R'')$, but it does belong to $c'(R'')$. Indeed, by Lemmas 7 and 11, this pair belong to $c'(R'')$ iff $Z_2 \subseteq (x{+}y{+}x_2{+}y_1)\downarrow_{R''\cup\preceq}$, and we have

$$\begin{aligned} & x{+}y{+}x_2{+}y_1 \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_1{+}x{+}y{+}y_1{+}x_2 \qquad (Z_1{+}x{+}y{+}y_1\ R''\ x{+}y{+}y_1) \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_1{+}X_1{+}y{+}y_1{+}x_2 = Z_1{+}X_2{+}y{+}y_1 \quad (x_1 \preceq z_1) \\ \rightsquigarrow_{R''\cup\preceq}\ & Z_2{+}X_2{+}y{+}y_1{+}x_2 \qquad (Z_2{+}X_2{+}y\ R''\ X_2{+}y) \end{aligned}$$

On the contrary, AC' is not able to exploit similarity in this case, and it behaves like AC: both of them compute the same antichain T_z as in the example from Section 4.2.4, which has $2^{n+1}-1$ elements.

In fact, even when considering inclusion of disjoint automata, the use of similarity tends to virtually merge states, so that HKC' can use the up to transitivity technique which AC and AC' lack.

5.4 A short recap

Figure 7 summarises the relationship amongst the presented algorithms, in the general case and in the special case of language inclusion of disjoint automata. In this diagram, an arrow X→Y (from an algorithm X to Y) means that (a) Y can explore less states than X, and (b) Y can mimic X, i.e., the proof technique of Y is at least as powerful as the one of X. (The labels on the arrows point to the sections showing these relations; unlabelled arrows are not illustrated in this paper, they are easily inferred from what we have shown.)

6. Experimental assessment

To get an intuition of the average behaviour of HKC on various NFA, and to compare it with HK and AC, we provide some benchmarks on random automata and on automata obtained from model-checking problems. In both cases, we conduct the experiments on a MacBook pro 2.4GHz Intel Core i7, with 4GB of memory, running OS X Lion (10.7.4). We use our OCaml implementation for HK, HKC, and HKC' [6], and the libvata C++ library for AC and AC' [20]. (To our knowledge, libvata is the most efficient implementation currently available for the antichain algorithms.)

6.1 Random automata

For a given size n, we generate a thousand random NFA with n states and two letters. According to [31], we use a linear transition density of 1.25 (which means that the expected out-degree of each state and with respect to each letter is 1.25): Tabakov and Vardi empirically showed that one statistically gets more challenging NFA with this particular value. We generate NFA without accepting states: by doing so, we make sure that the algorithms never encounter a counter-example, so that they always continue until they find a (bi)simulation up to: these runs correspond to their worst cases for all possible choices of accepting states for the given NFA.[2]

We run all algorithms on these NFA, starting from two distinct singleton sets, to measure the required time and the number of processed pairs: for HK, HKC, and HKC', this is the number of pairs put into the bisimulation up to (R); for AC and AC', this is the number of pairs inserted into the antichain. The timings for HKC' and AC' do not include the time required to compute similarity.

We report the median values (50%), the last deciles (90%), the last percentiles (99%), and the maximum values (100%) in Table 1. For instance, for $n = 70$, 90% of the examples require less than 155ms with HK; equivalently, 10% of the examples require more than 155ms. (For a few tests, libvata ran out of memory, whence the ∞ symbols in the table.) We also plotted on Figure 8 the distribution of the number of processed pairs when $n = 100$.

HKC and AC are several orders of magnitude better than HK, and HKC is usually two to ten times faster than AC. Moreover, for the first four lines, HKC is much more predictable than AC, i.e., the last percentiles and maximal values are of the same order as the median value. (AC seems to become more predictable for larger values of n.) The same relative behaviour can be observed between HKC' and AC'; moreover, HKC alone is apparently faster than AC'.

Also recall that the size of the relations generated by HK is a lower bound for the number of accessible states of the determinised NFA (Lemma 6 (2)); one can thus see in Table 1 that HKC usually explores an extremely small portion of these DFA (e.g., less than one per thousand for $n = 100$). The last column reports the median size of the minimal DFA for the corresponding parameters, as given in [31]. HK usually explores much many states than what would be necessary with a minimal DFA, while HKC and AC need much less.

6.2 Automata from model-checking

Checking language inclusion of NFA can be useful for model-checking, where one sometimes has to compute a sequence of NFA

[2] To get this behaviour for AC and AC', we actually had to trick libvata, which otherwise starts by removing non-coaccessible states, and thus reduces any of these NFA to the empty one.

hal-00639716, version 5 - 11 Jul 2012

hal-00639716, version 5 - 11 Jul 2012

		required time (seconds)				number of processed pairs				mDFA size
$n = \|S\|$	algo.	50%	90%	99%	100%	50%	90%	99%	100%	50%
50	HK	0.007	0.022	0.050	0.119	2511	6299	12506	25272	
	AC	0.002	0.003	0.142	1.083	112	245	2130	5208	
	HKC	0.000	0.000	0.000	0.000	21	26	32	63	∼1000
	AC'	0.002	0.002	0.038	0.211	79	131	1098	1926	
	HKC'	0.000	0.000	0.000	0.000	18	23	28	58	
70	HK	0.047	0.155	0.413	0.740	10479	28186	58782	87055	
	AC	0.002	0.003	1.492	4.163	150	285	8383	15575	
	HKC	0.000	0.000	0.000	0.000	27	34	40	49	∼6000
	AC'	0.002	0.003	0.320	0.884	110	172	3017	6096	
	HKC'	0.000	0.000	0.000	0.000	23	29	36	44	
100	HK	0.373	1.207	3.435	5.660	58454	164857	361227	471727	
	AC	0.003	0.004	3.214	36.990	204	298	13801	48059	
	HKC	0.000	0.000	0.000	0.001	36	44	54	70	∼30000
	AC'	0.003	0.004	0.738	6.966	152	211	4087	18455	
	HKC'	0.000	0.000	0.000	0.001	31	39	46	64	
300	AC	0.009	0.010	0.028	0.750	562	622	2232	14655	
	HKC	0.001	0.002	0.003	0.009	86	104	118	132	–
	AC'	0.012	0.013	0.022	0.970	433	484	920	14160	
	HKC'	0.001	0.001	0.002	0.006	76	91	104	116	
500	AC	0.014	0.015	0.039	∞	918	986	2571	∞	
	HKC	0.002	0.005	0.008	0.018	130	154	176	193	–
	AC'	0.025	0.028	0.042	∞	710	772	1182	∞	
	HKC'	0.002	0.004	0.007	0.013	115	136	154	169	
1000	AC	0.029	0.031	0.038	∞	1808	1878	2282	∞	
	HKC	0.007	0.022	0.055	0.093	228	271	304	337	–
	AC'	0.074	0.080	0.092	∞	1409	1488	1647	∞	
	HKC'	0.008	0.019	0.041	0.077	202	238	265	299	

Table 1. Running the five presented algorithms to check language equivalence on random NFA with two letters.

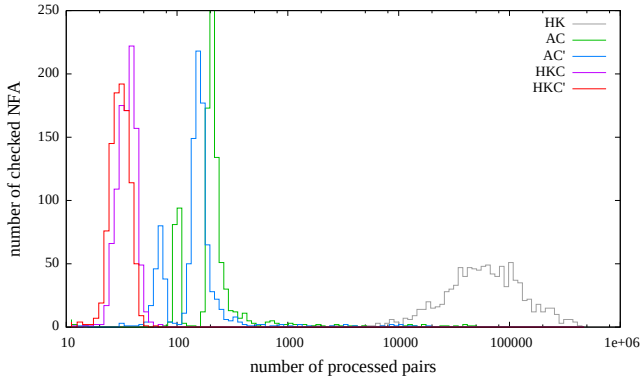


Figure 8. Distributions of the number of processed pairs, for the 1000 NFA with 100 states and 2 letters from Table 1.

by iteratively applying a transducer, until a fixpoint is reached [7]. To know that the fixpoint is reached, one typically has to check whether an NFA is contained in another one.

Abdulla et al. [1] use such benchmarks to test their algorithm (`AC'`) against the plain antichain algorithm (`AC` [33]). We reuse them to test `HKC'` against `AC'` in a concrete scenario. We take the sequences of automata kindly provided by L. Holik, which roughly corresponds to those used in [1] and which come from the model checking of various programs (the bakery algorithm, bubble sort, and a producer-consumer system). For all these sequences, we check the inclusions of consecutive pairs, in both directions. We separate the results into those for which a counter-example is found, and those for which the inclusion holds. We skip the trivial inclusions which hold by similarity ($\preceq^{\forall\exists}$), and for which both `HKC'` and `AC'` stop immediately.

The results are given in Table 2. Even though these are inclusions of disjoint automata, `HKC'` is faster than `AC'` on these examples: up to transitivity can be exploited thanks to the similarity pairs, and larger parts of the determinised NFA can be skipped.

7. Related work

A similar notion of bisimulation up to congruence has already been used to obtain decidability and complexity results about context-free processes, under the name of *self-bisimulations*. Caucal [10] introduced this concept to give a shorter and nicer proof of the result by Baeten et al. [4]: bisimilarity is decidable for normed context-free processes. Christensen et al [11] then generalised the result to all context-free processes, also by using self-bisimulations. Hirshfeld et al. [14] used a refinement of this notion to get a polynomial algorithm for bisimilarity in the normed case.

There are two main differences with the ideas we presented here. First, the above papers focus on bisimilarity rather than language equivalence (recall that although we use bisimulation relations, we check language equivalence since we work on the determinised NFA—Remark 3). Second, we consider a notion of bisimulation up to congruence where the congruence is taken with respect to non-determinism (union of sets of states). Self-bisimulations are also bisimulations up to congruence, but the congruence is taken with respect to word concatenation. We cannot consider this operation in our setting since we do not have the corresponding monoid structure in plain NFA.

Other approaches, that are independent from the algebraic structure (e.g., monoids or semi-lattices) and the behavioural equivalence (e.g., bisimilarity or language equivalence) are shown in [5, 21, 22, 26]. These propose very general frameworks into which our up to congruence technique fits as a very special case. To our knowledge, bisimulation up to congruence has never been proposed as a technique for proving language equivalence of NFA.

hal-00639716, version 5 - 11 Jul 2012

		required time (seconds)				number of processed pairs				number of tests
result	algo.	50%	90%	99%	100%	50%	90%	99%	100%	
counter-example	AC'	0.012	0.107	1.047	1.134	23	247	598	1352	518
	HKC'	0.001	0.005	0.025	0.383	11	24	112	290	
inclusion holds	AC'	0.079	0.795	1.457	1.480	149	733	1854	3087	178
	HKC'	0.015	0.165	0.340	0.345	61	695	1076	1076	

Table 2. Running HKC' and AC' to test language inclusion of disjoint NFA generated from model-checking.

8. Conclusions and future work

We showed that the standard algorithm by Hopcroft and Karp for checking language equivalence of DFA relies on a bisimulation up to equivalence proof technique; this allowed us to design a new algorithm (HKC) for the non-deterministic case, where we exploit a novel technique called up to congruence.

We then compared HKC to the recently introduced antichain algorithms [33] (AC): when checking the inclusion of disjoint automata, the two algorithms are equivalent, in all the other cases HKC is more efficient since it can use transitivity to prune a larger portion of the state-space.

The difference between these two approaches becomes even more striking when considering some optimisation exploiting similarity. Indeed, as nicely shown with AC' [1], the antichains approach can widely benefit from the knowledge one gets by first computing similarity. Inspired by this work, we showed that both our proof technique (bisimulation up to congruence) and our algorithm (HKC) can be easily modified to exploit similarity. The resulting algorithm (HKC') is now more efficient than AC' even for checking language inclusion of disjoint automata.

We provided concrete examples where HKC and HKC' are exponentially faster than AC and AC' (Sections 4.2.4 and 5.3) and we proved that the coinductive techniques underlying the formers are at least as powerful as those exploited by the latters (Propositions 5 and 7). We finally compared the algorithms experimentally, by running them on both randomly generated automata, and automata resulting from model checking problems. It appears that for these examples, HKC and HKC' perform better than AC and AC'.

Finally note that our implementation of the presented algorithms is available online [6], together with an applet making it possible to test them on user-provided examples.

As future work, we plan to extend our approach to tree automata. In particular, it seems promising to investigate if further up-to techniques can be defined for regular tree expressions. For instance, the algorithms proposed in [3, 18] exploit some optimisation which suggest us coinductive up-to techniques.

References

[1] P. A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In *Proc. TACAS*, vol. 6015 of *LNCS*, pages 158–174. Springer, 2010.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] A. Aiken and B. R. Murphy. Implementing regular tree expressions. In *FPCA*, vol. 523 of *LNCS*, pages 427–447. Springer, 1991.

[4] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. In *Proc. PARLE (II)*, vol. 259 of *LNCS*, pages 94–111. Springer, 1987.

[5] F. Bartels. *On generalized coinduction and probabilistic specification formats*. PhD thesis, Vrije Universiteit Amsterdam, 2004.

[6] F. Bonchi and D. Pous. Web appendix for this paper. http://perso.ens-lyon.fr/damien.pous/hknt, 2012.

[7] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proc. CAV*, vol. 3114 of *LNCS*. Springer, 2004.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[9] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In *Mathematical Theory of Automata*, vol. 12(6), pages 529–561. Polytechnic Press, NY, 1962.

[10] D. Caucal. Graphes canoniques de graphes algébriques. *ITA*, 24:339–352, 1990.

[11] S. Christensen, H. Hüttel, and C. Stirling. Bisimulation equivalence is decidable for all context-free processes. *Information and Computation*, 121(2):143–148, 1995.

[12] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1 (2/3):251–273, 1992.

[13] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS*, pages 453–462. IEEE Computer Society, 1995.

[14] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimilarity of normed context-free processes. *Theoretical Computer Science*, 158(1&2):143–159, 1996.

[15] J. E. Hopcroft. An n log n algorithm for minimizing in a finite automaton. In *Proc. International Symposium of Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[16] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. TR 114, Cornell Univ., December 1971.

[17] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[18] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[19] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In *Proc. STOC*, pages 264–274. ACM, 1992.

[20] O. Lengál, J. Simácek, and T. Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *TACAS*, vol. 7214 of *LNCS*, pages 79–94. Springer, 2012.

[21] M. Lenisa. From set-theoretic coinduction to coalgebraic coinduction: some results, some problems. *ENTCS*, 19:2–22, 1999.

[22] D. Lucanu and G. Rosu. Circular coinduction with special contexts. In *Proc. ICFEM*, vol. 5885 of *LNCS*, pages 639–659. Springer, 2009.

[23] A. Meyer and L. J. Stockmeyer. Word problems requiring exponential time. In *Proc. STOC*, pages 1–9. ACM, 1973.

[24] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[25] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I/II. *Information and Computation*, 100(1):1–77, 1992.

[26] D. Pous. Complete lattices and up-to techniques. In *Proc. APLAS*, vol. 4807 of *LNCS*, pages 351–366. Springer, 2007.

[27] J. Rutten. Automata and coinduction (an exercise in coalgebra). In *Proc. CONCUR*, vol. 1466 of *LNCS*, pages 194–218. Springer, 1998.

[28] D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.

[29] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[30] A. Silva, F. Bonchi, M. Bonsangue, and J. Rutten. Generalizing the powerset construction, coalgebraically. In *Proc. FSTTCS*, vol. 8 of *LIPIcs*, pages 272–283. Leibniz-Zentrum fuer Informatik, 2010.

hal-00639716, version 5 - 11 Jul 2012

[31] D. Tabakov and M. Vardi. Experimental evaluation of classical automata constructions. In *Proc. LPAR*, vol. 3835 of *LNCS*, pages 396–411. Springer, 2005.

[32] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291, 1997.

[33] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Proc. CAV*, vol. 4144 of *LNCS*, pages 17–30. Springer, 2006.

A. Smallest bisimulation and compositionality

In this appendix, we show some (unrelated) properties that have been discussed through the paper, but never formally stated.

The first property concerns the relation computed by $\texttt{Naive}(x, y)$. The following proposition shows that it is the *smallest bisimulation* relating x and y.

Proposition 8. *Let x and y be two states of a DFA. Let $R_{\texttt{Naive}}$ be the relation built by $\texttt{Naive}(x, y)$. If $\texttt{Naive}(x, y) = true$, then $R_{\texttt{Naive}}$ is the smallest bisimulation relating x and y, i.e., $R_{\texttt{Naive}} \subseteq R$, for all bisimulations R such that $(x, y) \in R$.*

Proof. We have already shown in Proposition 2 that $R_{\texttt{Naive}}$ is a bisimulation. We need to prove that it is the smallest. Let R be a bisimulation such that $(x, y) \in R$. For all words $w \in A^*$ and pair of states (x', y') such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$, it must hold that $(x', y') \in R$ (by definition of bisimulation).

By construction, for all $(x', y') \in R_{\texttt{Naive}}$ there exists a word $w \in A^*$, such that $x \xrightarrow{w} x'$ and $y \xrightarrow{w} y'$. Therefore all the pairs in $R_{\texttt{Naive}}$ must be also in R, that is $R_{\texttt{Naive}} \subseteq R$. □

The second property is

$$[\![X + Y]\!] = [\![X]\!] + [\![Y]\!] \ ,$$

which we have used in the Introduction to give an intuition of bisimulation up to context and to show that the problem of language inclusion can be reduced to language equivalence. We believe that this property is interesting, since it follows from the categorical observation made in [30] that determinised NFA are bialgebras [32], like CCS processes. For this reason, we prove here that $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism.

Theorem 7. *Let (S, o, t) be a non-deterministic automaton and $(\mathcal{P}(S), o^\sharp, t^\sharp)$ be the corresponding deterministic automaton obtained through the powerset construction. The function $[\![-]\!] \colon \mathcal{P}(S) \to 2^{A^*}$ is a semi-lattice homomorphism, that is, for all $X_1, X_2 \in \mathcal{P}(S)$,*

$$[\![X_1 + X_2]\!] = [\![X_1]\!] + [\![X_2]\!] \quad \textit{and} \quad [\![0]\!] = 0 \, .$$

Proof. We prove that for all words $w \in A^*$, $[\![X_1 + X_2]\!](w) = [\![X_1]\!](w) + [\![X_2]\!](w)$, by induction on w.

- for ϵ, we have:
$$\begin{aligned}[\![X_1 + X_2]\!](\epsilon) &= o^\sharp(X_1 + X_2) \\ &= o^\sharp(X_1) + o^\sharp(X_2) = [\![X_1]\!](\epsilon) + [\![X_2]\!](\epsilon) \, .\end{aligned}$$
- for $a \cdot w$, we have:
$$\begin{aligned}&[\![X_1 + X_2]\!](a \cdot w) \\ &= [\![t_a^\sharp(X_1 + X_2)]\!](w) && \text{(by definition)} \\ &= [\![t_a^\sharp(X_1) + t_a^\sharp(X_2)]\!](w) && \text{(by definition)} \\ &= [\![t_a^\sharp(X_1)]\!](w) + [\![t_a^\sharp(X_2)]\!](w) && \text{(by induction hypothesis)} \\ &= [\![X_1]\!](a \cdot w) + [\![X_2]\!](a \cdot w) \, . && \text{(by definition)}\end{aligned}$$

For the second part, we prove that for all words $w \in A^*$, $[\![0]\!](w) = 0$, again by induction on w. *Base case:* $[\![0]\!](\epsilon) = o^\sharp(0) = 0$. *Inductive case:* $[\![0]\!](a \cdot w) = [\![t_a^\sharp(0)]\!](w) = [\![0]\!](w)$ that by induction hypothesis is 0. □

B. Proofs of Section 2

Proposition 1. Two states are language equivalent iff there exists a bisimulation that relates them.

Proof. Let $R_{[\![-]\!]}$ be the relation $\{(x, y) \mid [\![x]\!] = [\![y]\!]\}$. We prove that $R_{[\![-]\!]}$ is a bisimulation. If $x \ R_{[\![-]\!]} \ y$, then $o(x) = [\![x]\!](\epsilon) = [\![y]\!](\epsilon) = o(y)$. Moreover, for all $a \in A$ and $w \in A^*$, $[\![t_a(x)]\!](w) = [\![x]\!](a \cdot w) = [\![y]\!](a \cdot w) = [\![t_a(y)]\!](w)$ that means $[\![t_a(x)]\!] = [\![t_a(y)]\!]$, that is $t_a(x) \ R_{[\![-]\!]} \ t_a(y)$.

We now prove the other direction. Let R be a bisimulation. We want to prove that $x \ R \ y$ entails $[\![x]\!] = [\![y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) = [\![y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) = o(y) = [\![y]\!](\epsilon)$. For $w = a \cdot w'$, since R is a bisimulation, we have $t_a(x) \ R \ t_a(y)$ and thus $[\![t_a(x)]\!](w') = [\![t_a(y)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a \cdot w') = [\![t_a(x)]\!](w')$ and $[\![y]\!](a \cdot w') = [\![t_a(y)]\!](w')$. □

Lemma 1. The following functions are compatible:

id: the identity function;

$f \circ g$: the composition of compatible functions f and g;

$\bigcup F$: the pointwise union of an arbitrary family F of compatible functions: $\bigcup F(R) = \bigcup_{f \in F} f(R)$;

f^ω: the (omega) iteration of a compatible function f.

Proof. The first two points are straightforward;

For the third one, assume that F is a family of compatible functions. Suppose that $R \rightarrowtail R'$; for all $f \in F$, we have $f(R) \rightarrowtail f(R')$ so that $\bigcup_{f \in F} f(R) \rightarrowtail \bigcup_{f \in F} f(R')$.

For the last one, assume that f is compatible; for all n, f^n is compatible because (a) $f^0 = id$ is compatible (by the first point) and (b) $f^{n+1} = f \circ f^n$ is compatible (by the second point and induction hypothesis). By definition $f^\omega = \bigcup_n f^n$ and thus, by the third point, f^ω is compatible. □

Lemma 2. The following functions are compatible:

- the constant reflexive function: $r(R) = \{(x, x) \mid \forall x \in S\}$;
- the converse function: $s(R) = \{(y, x) \mid x \ R \ y\}$;
- the squaring function: $t(R) = \{(x, z) \mid \exists y, x \ R \ y \ R \ z\}$.

Proof. r: observe that the identity relation $Id = \{(x, x) \mid \forall x \in S\}$ is always a bisimulation, i.e., $Id \rightarrowtail Id$. Thus for all R, R' $r(R) = Id \rightarrowtail Id = r(R')$.

s: observe that the definition of progression is completely symmetric. Therefore, if $R \rightarrowtail R'$, then $s(R) \rightarrowtail s(R')$.

t: assume that $R \rightarrowtail R'$. For each $(x, z) \in t(R)$, there exists y such that $(x, y) \in R$ and $(y, z) \in R$. By assumption, (1) $o'(x) = o'(y) = o'(z)$ and (2) for all $a \in A$, $t'_a(x) \ R' \ t'_a(y) \ R' \ t'_a(z)$, that is $t'_a(x) \ t(R') \ t'_a(z)$.

□

hal-00639716, version 5 - 11 Jul 2012

C. Proofs of Section 3

Lemma 4. For all relations R, the relation $\rightsquigarrow_R$ is convergent.

Proof. We have that $Z \rightsquigarrow_R Z'$ implies $|Z'| > |Z|$, where $|X|$ denotes the cardinality of the set X (note that $\rightsquigarrow_R$ is irreflexive). Since $|Z'|$ is bounded by $|S|$, the number of states of the NFA, the relation $\rightsquigarrow_R$ is strongly normalising. We can also check that whenever $Z \rightsquigarrow_R Z_1$ and $Z \rightsquigarrow_R Z_2$, either $Z_1 = Z_2$ or there is some Z' such that $Z_1 \rightsquigarrow_R Z'$ and $Z_2 \rightsquigarrow_R Z'$. Therefore, $\rightsquigarrow_R$ is convergent. □

Lemma 13. *The relation $\rightsquigarrow_R$ is contained in $c(R)$.*

Proof. If $Z \rightsquigarrow_R Z'$ then there exists $(X, Y) \in (s \cup \mathrm{id})(R)$ such that $Z = Z + X$ and $Z' = Z + Y$. Therefore $Z\ c(R)\ Z'$ and, thus, $\rightsquigarrow_R$ is contained in $c(R)$. □

Lemma 14. *Let $X, Y \in \mathcal{P}(S)$, we have $(X + Y)\downarrow_R = (X\downarrow_R + Y\downarrow_R)\downarrow_R$.*

Proof. Follows from confluence (Lemma 4) and from the fact that for all Z, Z', U, $Z \rightsquigarrow_R Z'$ entails $U + Z \rightsquigarrow_R^{=} U + Z'$. □

Theorem 3. For all relations R, and for all $X, Y \in \mathcal{P}(S)$, we have $X\downarrow_R = Y\downarrow_R$ iff $(X, Y) \in c(R)$.

Proof. From right to left. We proceed by induction on the derivation of $(X, Y) \in c(R)$. The cases for rules r, s, and t are straightforward. For rule id, suppose that $X\ R\ Y$, we have to show $X\downarrow_R = Y\downarrow_R$:

- if $X = Y$, we are done;
- if $X \subsetneq Y$, then $X \rightsquigarrow_R X + Y = Y$;
- if $Y \subsetneq X$, then $Y \rightsquigarrow_R X + Y = X$;
- if neither $Y \subseteq X$ nor $X \subseteq Y$, then $X, Y \rightsquigarrow_R X + Y$.

(In the last three cases, we conclude by confluence—Lemma 4.)
For rule u, suppose by induction that $X_i\downarrow_R = Y_i\downarrow_R$ for $i \in 1, 2$; we have to show that $(X_1 + Y_1)\downarrow_R = (X_2 + Y_2)\downarrow_R$. This follows by Lemma 14.

From left to right. By Lemma 13, we have $X\ c(R)\ X\downarrow_R$ for any set X, so that $X\ c(R)\ X\downarrow_R = Y\downarrow_R\ c(R)\ Y$. □

Lemma 5. The three algorithms require at most $1+v\cdot|R|$ iterations, where $|R|$ is the size of the produced relation; moreover, this bound is reached whenever they return true.

Proof. At each iteration, one pair is extracted from $todo$. The latter contains one pair before entering the loop and v pairs are added to it every time that a pair is added to R. □

Lemma 15. *Let x and y be two states of a DFA. Let $R_{\mathtt{Naive}}$ and $R_{\mathtt{HK}}$ be relations computed by $\mathtt{Naive}(x, y)$ and $\mathtt{HK}(x, y)$, respectively. If $\mathtt{Naive}(x, y) = \mathtt{HK}(x, y) = true$, then $e(R_{\mathtt{Naive}}) = e(R_{\mathtt{HK}})$.*

Proof. By the proof of Proposition 3, $e^\omega(R_{\mathtt{HK}})$ is a bisimulation. Since e is idempotent, we have $e^\omega = e$ and thus $e(R_{\mathtt{HK}})$ is a bisimulation; we can thus deduce by Proposition 8 that $R_{\mathtt{Naive}} \subseteq e(R_{\mathtt{HK}})$. Moreover, by definition of the algorithms, we have $R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}}$. Summarising,

$$R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}} \subseteq e(R_{\mathtt{HK}})$$

It follows that $e(R_{\mathtt{HK}}) = e(R_{\mathtt{Naive}})$, e being monotonic and idempotent. □

Lemma 6. Let $R_{\mathtt{Naive}}$, $R_{\mathtt{HK}}$, and $R_{\mathtt{HKC}}$ denote the relations produced by the three algorithms. We have

$$|R_{\mathtt{HKC}}|, |R_{\mathtt{HK}}| \leq m \qquad |R_{\mathtt{Naive}}| \leq m^2\ , \tag{2}$$

where $m \leq 2^n$ is the number of accessible states in the determinised NFA and n is the number of states of the NFA. If the algorithms returned true, we moreover have

$$|R_{\mathtt{HKC}}| \leq |R_{\mathtt{HK}}| \leq |R_{\mathtt{Naive}}|\ . \tag{3}$$

Proof. For the first point, let PS denote the set of (determinised NFA) states accessible from the two starting states, so that $m = |PS| \leq 2^n$. Since $R_{\mathtt{Naive}} \subseteq PS \times PS$, we deduce $|R_{\mathtt{Naive}}| \leq m^2$. Since each pair added to $R_{\mathtt{HK}}$ merges two distinct equivalence classes in $e(R_{\mathtt{HK}})$, we necessarily have $|R_{\mathtt{HK}}| \leq m$ (the largest partition of PS has exactly m singletons). Similarly, each pair added to $R_{\mathtt{HKC}}$ merges *at least* two distinct equivalence classes in $c(R_{\mathtt{HK}})$, so that we also have $|R_{\mathtt{HKC}}| \leq m$.

For the second point, $|R_{\mathtt{HK}}| \leq |R_{\mathtt{Naive}}|$ follows from the fact that $R_{\mathtt{HK}} \subseteq R_{\mathtt{Naive}}$, by definition of the algorithms. The other inequality is less obvious.

By construction, $R_{\mathtt{HKC}} \subseteq R_{\mathtt{Naive}}$ and, since e is monotonic, $e(R_{\mathtt{HKC}}) \subseteq e(R_{\mathtt{Naive}}) = e(R_{\mathtt{HK}})$ (the latter equality is given by Proposition 15). In particular, there are more equivalence classes in $e(R_{\mathtt{HKC}})$ than in $e(R_{\mathtt{HK}})$; using the same argument as above, we deduce that $|R_{\mathtt{HKC}}| \leq |R_{\mathtt{HK}}|$. □

Lemma 8. Let X, Y be two sets of states; let $R_\subseteq$ and $R_\supseteq$ be the relations computed by $\mathtt{HKC}(X{+}Y, Y)$ and $\mathtt{HKC}(X{+}Y, X)$, respectively. If $R_\subseteq$ and $R_\supseteq$ are bisimulations up to congruence, then the following relation is a bisimulation up to congruence:

$$R_= = \{(X', Y') \mid (X'{+}Y', Y') \in R_\subseteq \text{ or } (X'{+}Y', X') \in R_\supseteq\}.$$

Proof. Let $(X', Y') \in R_=$ and suppose that $(X'{+}Y', Y') \in R_\subseteq$ (the other case is symmetric).

First notice that all pairs in $R_\supseteq$ necessarily have the shape $(t_w^\sharp(X{+}Y),\ t_w^\sharp(X))$, for some word w. Since $R_\supseteq$ is a bisimulation up to congruence, $c(R_\supseteq)$ is a bisimulation. Since $(X{+}Y, X) \in c(R_\supseteq)$ then, for all words w, $(t_w^\sharp(X{+}Y), t_w^\sharp(X)) \in c(R_\supseteq)$ and thus $(X'{+}Y', X') \in c(R_\supseteq)$ (we have $X' = t_w^\sharp(X)$ and $Y' = t_w^\sharp(Y)$ for some word w).

Since $c(R_\subseteq)$ and $c(R_\supseteq)$ are bisimulations containing $(X'{+}Y', Y')$ and $(X'{+}Y', X')$, it holds that:

1. $o^\sharp(X') = o^\sharp(X' + Y') = o^\sharp(Y')$;
2. for all a, $t_a^\sharp(X' + Y')\ c(R_\supseteq)\ t_a^\sharp(X')$ and $t_a^\sharp(X' + Y')\ c(R_\subseteq)\ t_a^\sharp(Y')$.

By Lemma 7, $t_a^\sharp(Y') \subseteq t_a^\sharp(X')\downarrow_{R_\supseteq}$ and $X' \subseteq t_a^\sharp(Y')\downarrow_{R_\subseteq}$ and since all the rewriting rules for $R_\subseteq$ and $R_\supseteq$ are also rewriting rules for $R_=$, then $t_a^\sharp(Y') \subseteq t_a^\sharp(X')\downarrow_{R_=}$ and $t_a^\sharp(X') \subseteq t_a^\sharp(Y')\downarrow_{R_=}$. By the first observation in the proof of Lemma 7, this means that $t_a^\sharp(X')\ c(R_=)\ t_a^\sharp(Y')$. □

D. Proofs of Section 4

Proposition 4. For all sets X, Y, we have $[\![X]\!] \subseteq [\![Y]\!]$ iff there exists a simulation T such that for all $x \in X$, $x\ T\ Y$.

hal-00639716, version 5 - 11 Jul 2012

Proof. Let $T_{[\![-]\!]}$ be the relation $\{(x, Y) \mid [\![x]\!] \subseteq [\![Y]\!]\}$. We prove that $T_{[\![-]\!]}$ is a simulation. If $x \ T_{[\![-]\!]} \ Y$, then $o(x) = [\![x]\!](\epsilon) \leq [\![Y]\!](\epsilon) = o^\sharp(Y)$. Moreover, for all $a \in A$ $x' \in t_a(x)$ and $w \in A^*$, $[\![x']\!](w) \subseteq [\![x]\!](a \cdot w) \subseteq [\![Y]\!](a \cdot w) = [\![t_a^\sharp(Y)]\!](w)$ that means $[\![x']\!] \subseteq [\![t_a^\sharp(Y)]\!]$, that is $t_a(x) \ T_{[\![-]\!]} \ t_a^\sharp(Y)$.

We now prove the other direction. Let T be a simulation. We want to prove that $x \ T \ Y$ entails $[\![x]\!] \subseteq [\![Y]\!]$, i.e., for all $w \in A^*$, $[\![x]\!](w) \leq [\![Y]\!](w)$. We proceed by induction on w. For $w = \epsilon$, we have $[\![x]\!](\epsilon) = o(x) \leq o^\sharp(Y) = [\![Y]\!](\epsilon)$. For $w = a \cdot w'$, since T is a simulation, we have $t_a(x) \ T \ t_a^\sharp(Y)$ and thus $[\![t_a(x)]\!](w') \leq [\![t_a^\sharp(Y)]\!](w')$ by induction. This allows us to conclude since $[\![x]\!](a \cdot w') = [\![t_a(x)]\!](w')$ and $[\![y]\!](a \cdot w') = [\![t_a^\sharp(y)]\!](w')$. □

Definition 10. *A function $f : \mathcal{P}(S \times \mathcal{P}(S)) \to \mathcal{P}(S \times \mathcal{P}(S))$ is* s-compatible *if is monotone and for all relations $T, T' \subseteq S \times \mathcal{P}(S)$, $T \rightarrowtail_s T'$ entails $f(T) \rightarrowtail_s f(T')$.*

Lemma 16. *Any simulation T up to an s-compatible function f ($T \rightarrowtail_s f(T)$) is contained in a simulation, namely $f^\omega(T)$.*

Proof. Same proof as for Proposition 3. □

Lemma 17. *The upward closure function $\uparrow$ is s-compatible.*

Proof. We assume that $T \rightarrowtail_s T'$ and we prove that $\uparrow T \rightarrowtail_s \uparrow T'$. If $x \uparrow T \ Y$, then $\exists Y' \subseteq Y$ such that $x \ T \ Y'$. Since $Y' \subseteq Y$, $o^\sharp(Y') \leq o^\sharp(Y)$ and $t_a^\sharp(Y') \subseteq t_a^\sharp(Y)$ for all $a \in A$. Since $T \rightarrowtail_s T'$ and $x \ T \ Y'$, then $o(x) \leq o^\sharp(Y') \leq o^\sharp(Y)$ and $t_a(x) \uparrow T' \ t_a^\sharp(Y)$ for all $a \in A$. □

Theorem 4. Any simulation up to $\uparrow$ is contained in a simulation.

Proof. By Lemmas 16 and 17. □

Lemma 18. *The relation*

$$\begin{aligned} R' = &\{(x + y,\ z)\} \\ &+ \{(x + Y_i + y_{i+1},\ Z_{i+1}) \mid i < n\} \\ &+ \{(x + Y_i + x_{i+1},\ Z_{i+1}) \mid i < n\} \end{aligned}$$

is a bisimulation up to congruence for the NFA in Fig. 5.

Proof. First notice that

$$X_1 + y \quad c(R') \quad x + Y_1 \quad c(R') \quad Z_1$$

We then consider each kind of pair of R' separately:

- (x, y): we have $o^\sharp(x + y) = 0 = o^\sharp(z)$ and $t_a^\sharp(x + y) = X_1 + y \ R' \ Z_1 = t_a^\sharp(z)$ and, similarly, $t_b^\sharp(x + y) = x + Y_1 \ R' \ Z_1 = t_b^\sharp(z)$.
- $(x + Y_i + y_{i+1},\ Z_{i+1})$: both members are accepting iff $i + 1 = n$; setting $j = \min(i + 2, n)$, we have

$$\begin{aligned} t_a^\sharp(x + Y_i + y_{i+1}) &= X_1 + y + y_2 + \cdots + y_j \\ &\quad c(R')\ x + Y_1 + y_2 + \cdots + y_j \\ &= x + Y_j \ R' \ Z_j = t_a^\sharp(Z_{i+1}) \end{aligned}$$

and

$$t_b^\sharp(x + Y_i + y_{i+1}) = x + Y_j \ R' \ Z_j = t_b^\sharp(Z_{i+1})$$

- $(x + Y_i + x_{i+1},\ Z_{i+1})$: both members are accepting iff $i + 1 = n$; if $i + 1 < n$ then we have:

$$\begin{aligned} t_a^\sharp(x + Y_i + x_{i+1}) &= X_1 + y + y_2 + \cdots + y_{i+1} + x_{i+2} \\ &\quad c(R')\ x + Y_1 + y_2 + \cdots + y_{i+1} + x_{i+2} \\ &= x + Y_{i+1} + x_{i+2} \\ &\quad R' \ Z_{i+2} = t_a^\sharp(Z_{i+1}) \end{aligned}$$

and

$$t_b^\sharp(x + Y_i + x_{i+1}) = x + Y_{i+1} + x_{i+2} \ R' \ Z_{i+2} = t_b^\sharp(Z_{i+1})$$

otherwise, i.e., $i + 1 = n$, notice that:

$$\begin{aligned} x + Y_n + x_n \ &c(R')\ Z_n + y_n \\ &c(R')\ x + Y_n + y_n = x + Y_n \\ &c(R')\ Z_n = t_a^\sharp(Z_n)\ , \end{aligned}$$

from which we deduce:

$$\begin{aligned} t_a^\sharp(x + Y_i + x_n) &= X_1 + y + y_2 + \cdots + y_n + x_n \\ &\quad c(R')\ x + Y_1 + y_2 + \cdots + y_n + x_n \\ &= x + Y_n + x_n \ c(R')\ t_a^\sharp(Z_n) \end{aligned}$$

and

$$t_b^\sharp(x + Y_i + x_n) = x + Y_n + x_n \ c(R')\ t_a^\sharp(Z_n)$$

□

E. Proofs of Section 5

Theorem 5. Any simulation up to $\curlywedge$ is contained in a simulation.

Proof. By Lemma 16, it suffices to show that $\curlywedge$ is s-compatible. Suppose that $T \rightarrowtail_s T'$, we have to show that $\curlywedge T \rightarrowtail_s \curlywedge T'$. Assume that $x \curlywedge T \ Y$.

- if $x \preceq^{\forall\exists} Y$ then $x \preceq y$ for some $y \in Y$. Therefore, we have $o(x) \leq o(y) \leq o^\sharp(Y)$ and for all $a \in A$, $x' \in t_a(x)$, we have some $y' \in t_a(y)$ with $x' \preceq y'$. Since $t_a(y) \subseteq t_a^\sharp(Y)$, we deduce $x' \preceq^{\forall\exists} t_a^\sharp(Y)$, and hence $x' \curlywedge T' \ t_a^\sharp(Y)$, as required.
- otherwise, we have some $(x', Y') \in T$ such that $(x', Y') \sqsubseteq^{\preceq} (x, Y)$, i.e., $x \preceq x'$ and $Y' \preceq^{\forall\exists} Y$. Since $T \rightarrowtail_s T'$, we have $o(x) \leq o(x') \leq o^\sharp(Y') \leq o^\sharp(Y)$. Now take some $x'' \in t_a(x)$, we have some $x''' \in t_a(x')$ with $x'' \preceq x'''$, and since $T \rightarrowtail_s T'$, we know $x''' \ T' \ t_a^\sharp(Y')$. It suffices to show that $t_a^\sharp(Y') \preceq^{\forall\exists} t_a^\sharp(Y)$ to conclude; this follows easily from $Y' \preceq^{\forall\exists} Y$ and from the definition of similarity. □

Lemma 11. For all relations R, $c'(R) = c(R \cup \overline{\preceq})$.

Proof. The inclusion $c(R \cup \overline{\preceq}) \subseteq c'(R)$ is trivial. For the other inclusion we take $d = r' \cup s \cup t \cup u \cup id$ and we prove by induction that for all natural numbers n, $d^n(R) \subseteq c(R \cup \overline{\preceq})$. For $n = 0$, $d^0(R) = R \subseteq c(R \cup \overline{\preceq})$. For $n + 1$, $d^{n+1}(R) = d(d^n(R))$. By induction hypothesis, $d^n(R) \subseteq c(R \cup \overline{\preceq})$ and, by monotonicity of d, $d(d^n(R)) \subseteq d(c(R \cup \overline{\preceq}))$. By definition of d, the latter is equal to $c(R \cup \overline{\preceq})$. □

Lemma 12. For all relations $T \subseteq S \times \mathcal{P}(S)$, $\widehat{\curlywedge T} \subseteq c'(\widehat{T})$.

hal-00639716, version 5 - 11 Jul 2012

Proof. If $(x+Y,Y) \in \widehat{\curlywedge T}$, then either (a) $x \preceq^{\forall\exists} Y$ or (b) there exists $x \preceq x'$ and $Y' \preceq^{\forall\exists} Y$ such that $(x',Y') \in T$. We have to show $(x+Y,Y) \in c'(\widehat{T})$, i.e., $(x+Y,Y) \in c(\widehat{T}+\overline{\preceq})$ by Lemma 11, that is $x \in Y{\downarrow_{\widehat{T}+\overline{\preceq}}}$ by Lemma 7. For (b), we have:

$$\begin{aligned} Y &\rightsquigarrow^{\star}_{\widehat{T}+\overline{\preceq}} Y+Y' && (Y' \preceq^{\forall\exists} Y)\\ &\rightsquigarrow_{\widehat{T}+\overline{\preceq}} Y+Y'+x' && ((x'+Y',Y') \in \widehat{T})\\ &\rightsquigarrow_{\widehat{T}+\overline{\preceq}} Y+Y'+x'+x && (x \preceq x') \end{aligned}$$

$x \in Y{\downarrow_{\widehat{T}+\overline{\preceq}}}$ follows by confluence (Lemma 4). For (a), we immediately have that $Y \rightsquigarrow_{\widehat{T}+\overline{\preceq}} Y+x$. □

Proposition 7. If T is a simulation up to $\curlywedge$, then $\widehat{T}$ is a bisimulation up to c'.

Proof. First observe that if $T \rightarrowtail_s T'$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{T'})$. Therefore, if $T \rightarrowtail_s \uparrow T$, then $\widehat{T} \rightarrowtail u^\omega(\widehat{\uparrow T})$. By Lemma 12, $\widehat{T} \rightarrowtail u^\omega(c'(\widehat{T})) = c'(\widehat{T})$. □

Lemma 19. *The relation*

$$\begin{aligned} R'' = &\{(z+x+y,\ x+y)\}\\ &\cup \{(Z_{i+1}+X_i+y+y_{i+1},\ X_i+y+y_{i+1}) \mid i<n\}\\ &\cup \{(Z_{i+1}+X_{i+1}+y,\ X_{i+1}+y) \mid i<n\}\ , \end{aligned}$$

is a bisimulation up to c' for the NFA in Figure 6.

Proof. Let X'_i be the set X_i without x_1 and note that $X_i \xrightarrow{a} X_{i+1}$ and $X_i \xrightarrow{b} X'_{i+1}$. First we observe that for all i,

$$X'_i+Y_1 \rightsquigarrow_{R''\cup\overline{\preceq}} X'_i+Y_1+Z_1 \rightsquigarrow_{R''\cup\overline{\preceq}} X'_i+Y_1+Z_1+x_1$$

where the first reduction is given by $(Z_1+X_0+y+y_1, X_0+y+y_1) \in R''$ and the second by $x_1 \preceq z_1$. Since $X'_i+x_1 = X_i$, then one can apply the third kind of pairs in R'', so that

$$X'_i+Y_1 \rightsquigarrow^*_{R''\cup\overline{\preceq}} X_i+Y_1+Z_i$$

that is $Z_i \subseteq (X'_i+Y_1){\downarrow_{R''\cup\overline{\preceq}}}$. By Lemmas 7 and 11, this means that

$$Z_i+X'_i+Y_1\ c'(R'')\ X'_i+Y_1 \tag{2}$$

If we moreover have y_{i+1}, we can apply the second kind of pair in R'' and obtain

$$X'_i+Y_1+y_{i+1} \rightsquigarrow^*_{R''\cup\overline{\preceq}} X_i+Y_1+Z_{i+1}+y_{i+1}$$

that is

$$Z_{i+1}+X'_i+Y_1+y_{i+1}\ c'(R'')\ X'_i+Y_1+y_{i+1} \tag{3}$$

With (2) and (3), it is easy to prove that R'' is a bisimulation up to c', by simply proceeding by cases:

- $(z+x+y,\ x+y)$: we have $o^\sharp(x+y+z) = 0 = o^\sharp(x+y)$ and $t^\sharp_a(x+y+z) = Z_1+X_1+y\ R''\ X_1+y = t^\sharp_a(x+y)$ and, similarly, $t^\sharp_b(x+y+z) = Z_1+x+Y_1\ R''\ x+Y_1 = t^\sharp_b(z)$.
- $(Z_{i+1}+X_i+y+y_{i+1},\ X_i+y+y_{i+1})$ and $i < n-1$: both members are not accepting;

$$\begin{aligned} t^\sharp_a(Z_{i+1}+X_i+y+y_{i+1}) &= Z_{i+2}+X_{i+1}+y+y_{i+2}\\ &R''\ X_{i+1}+y+y_{i+2}\\ &= t^\sharp_a(X_i+y+y_{i+1}) \end{aligned}$$

and

$$\begin{aligned} t^\sharp_b(Z_{i+1}+X_i+y+y_{i+1}) &= Z_{i+2}+X'_{i+1}+Y_1+y_{i+2}\\ &c'(R'')\ X'_{i+1}+Y_1+y_{i+2}\\ &= t^\sharp_b(X_i+y+y_{i+1}) \end{aligned}$$

- $(Z_n+X_{n-1}+y+y_n,\ X_{n-1}+y+y_n)$ and $i = n-1$: both members are accepting;

$$\begin{aligned} t^\sharp_a(Z_n+X_{n-1}+y+y_n) &= Z_n+X_n+y\\ &R''\ X_n+y\\ &= t^\sharp_a(X_{n-1}+y+y_n) \end{aligned}$$

and

$$\begin{aligned} t^\sharp_b(Z_n+X_{n-1}+y+y_n) &= Z_n+X'_n+Y_1\\ &c'(R'')\ X'_n+Y_1\\ &= t^\sharp_b(X_{n-1}+y+y_n) \end{aligned}$$

- $(Z_{i+1}+X_{i+1}+y,\ X_{i+1}+y)$ and $i < n-1$: both members are not accepting;

$$\begin{aligned} t^\sharp_a(Z_{i+1}+X_{i+1}+y) &= Z_{i+2}+X_{i+2}+y\\ &R''\ X_{i+2}+y\\ &= t^\sharp_a(X_{i+1}+y) \end{aligned}$$

and

$$\begin{aligned} t^\sharp_b(Z_{i+1}+X_{i+1}+y) &= Z_{i+2}+X'_{i+2}+Y_1\\ &c(R'')\ X'_{i+2}+Y_1\\ &= t^\sharp_b(X_{i+1}+y) \end{aligned}$$

- $(Z_n+X_n+y,\ X_n+y)$: both members are accepting; Moreover,

$$\begin{aligned} t^\sharp_a(Z_n+X_n+y) &= Z_n+X_n+y\\ &R''\ X_n+y = t^\sharp_a(X_n+y) \end{aligned}$$

and

$$\begin{aligned} t^\sharp_b(Z_n+X_n+y) &= Z_n+X'_n+Y_1\\ &c(R'')\ X'_n+Y_1\\ &= t^\sharp_b(X_n+y) \end{aligned}$$

The cases for the letter c are always trivial since $Z_i \xrightarrow{c} 0$. □

Concurrent Flexible Reversibility*

Ivan Lanese[1], Michael Lienhardt[2], Claudio Antares Mezzina[3], Alan Schmitt[4], and Jean-Bernard Stefani[4]

[1] Focus Team, University of Bologna/Inria, Italy lanese@cs.unibo.it
[2] PPS Laboratory, Paris Diderot University, France lienhard@cs.unibo.it
[3] SOA Unit, FBK, Trento, Italy mezzina@fbk.eu
[4] Inria, France alan.schmitt@inria.fr, jean-bernard.stefani@inria.fr

Abstract. Concurrent reversibility has been studied in different areas, such as biological or dependable distributed systems. However, only "rigid" reversibility has been considered, allowing to go back to a past state and restart the exact same computation, possibly leading to divergence. In this paper, we present CCSK, a concurrent calculus featuring *flexible reversibility*, allowing the specification of alternatives to a computation to be used upon rollback. Alternatives in CCSK are attached to messages. We show the robustness of this mechanism by encoding more complex idioms for specifying flexible reversibility, and we illustrate the benefits of our approach by encoding a calculus of communicating transactions.

1 Introduction

Reversible programs can be executed both in the standard, forward direction as well as in the backward direction, to go back to past states. Reversible programming is attracting much interest for its potential in several areas. For instance, chemical and biological reactions are typically bidirectional, and the direction of execution is fixed by environmental conditions such as temperature. Similarly, quantum computations are reversible as long as they are not observed. Reversibility is also used for backtracking in the exploration of a program state-space toward a solution, either as part of the design of the programming language as in Prolog, or to implement transactions. We are particularly interested in the use of reversibility for modeling and programming concurrent reliable systems. In this setting, the main idea is that in case of an error the program backtracks to a past state where the decisions leading to the error have not been taken yet, so that a new forward execution may avoid repeating the (same) error.

Reversibility has a non trivial interplay with concurrency. Understanding this interplay is fundamental in many of the areas above, e.g., for biological or reliable distributed systems, which are naturally concurrent. In the spirit of concurrency, independent threads of execution should be rolled-back independently, but causal dependencies between related threads should be taken into account.

* This work has been partially supported by the French National Research Agency (ANR), projects REVER ANR 11 INSE 007 and PiCoq ANR 10 BLAN 0305.

This form of reversibility, termed *causal consistent*, was first introduced by RCCS [11], a reversible variant of CCS. RCCS paved the way to the definition of reversible variants of more expressive concurrent calculi [8, 18, 20, 22]. This line of research considered rigid, uncontrolled, step-by-step reversibility. *Step-by-step* means that each single step can be undone, as opposed, e.g., to checkpointing where many steps are undone at once. *Uncontrolled* means that there is no hint as to when to go forward and when to go backward, and up to where. *Rigid* means that the execution of a forward step followed by the corresponding backward step leads back to the starting state, where an identical computation can restart.

While these works have been useful to understand the basics of concurrent reversibility in different settings, some means to *control* reversibility are required in practice. In the literature four different forms of control have been proposed: relating the direction of execution to some energy parameter [2], introducing irreversible actions [12], using an explicit rollback primitive [17], and using a superposition operator to control forward and backward execution [24].

With the exception of [24], these works were based on causal consistent, rigid reversibility. However, rigid reversibility may not always be the best choice. In the setting of reliable systems, for instance, rigid reversibility means that to recover from an error a past state is reached. From this past state the computation that lead to the error is still possible. If the error was due to a transient fault, retrying the same computation may be enough to succeed. If the failure was permanent, the program may redo the same error again and again.

Our goal is to overcome this limitation by providing the programmer with suitable linguistic constructs to specify what to do after a causal consistent backward computation. Such constructs can be used to ensure that new forward computations explore new possibilities. To this end, we build on our previous work on roll-π [17], a calculus where concurrent reversibility is controlled by the roll γ operator. Executing it reverses the action referred by γ together with all the dependent actions. Here, we propose a new calculus called croll-π, for compensating roll-π, as a framework for *flexible reversibility*. We attempt to keep croll-π as close as possible to roll-π while enabling many new possible applications. We thus simply replace roll-π communication messages $a\langle P\rangle$ by *messages with alternative* $a\langle P\rangle \div c\langle Q\rangle$. In forward computation, a message $a\langle P\rangle \div c\langle Q\rangle$ behaves exactly as $a\langle P\rangle$. However, if the interaction consuming it is reversed, the original message is not recreated—as would be the case with rigid reversibility—but the alternative $c\langle Q\rangle$ is released instead. Our rollback and alternative message primitives provide a simple form of reversibility control, which always respects the causal consistency of reverse computation. It contrasts with the fine-grained control provided by the superposition constructs in [24], where the execution of a CCS process can be constrained by a controller, possibly reversing given past actions in a way that is non-causally consistent.

Our contributions are as follows. We show that the simple addition of alternatives to roll-π greatly extends its expressive power. We describe how messages with alternative allow for programming different patterns for flexible reversibility. Then, we show that croll-π can be used to model the communicating transac-

tions of [13]. Notably, the tracking of causality of croll-π is more precise than the one in [13], thus allowing to improve on the original proposal by avoiding some spurious undo of actions. Additionally, we study some aspects of the behavioral theory of croll-π, including a context lemma for barbed congruence. This allows us to reason about croll-π programs, in particular to prove the correctness of the encodings of primitives for flexible reversibility and of the transactional calculus of [13]. Finally, we present an interpreter, written in Maude [10], for a small language based on croll-π.

Outline. Section 2 gives an informal introduction to croll-π. Section 3 defines the croll-π calculus, its reduction semantics, and it introduces the basics of its behavioral theory. Section 4 presents various croll-π idioms for flexible reversibility. Section 5 outlines the croll-π interpreter in Maude and a solution for the Eight Queens problem. Section 6 presents an encoding and an analysis of the Trans-CCS constructs from [13]. Section 7 concludes the paper with related work and a mention of future studies. The paper includes short proof sketches for the main results. We refer to the online technical report [16] for full proofs.

2 Informal Presentation

Rigid reversibility in roll-π. The croll-π calculus is a conservative extension of the roll-π calculus introduced in [17].[5] We briefly review the roll-π constructs before presenting the extension added by croll-π. Processes in roll-π are essentially processes of the asynchronous higher-order π-calculus [25], extended with a rollback primitive. Processes in roll-π cannot directly execute, only *configurations* can. A configuration is essentially a parallel composition of *tagged processes* along with *memories* tracking past interactions and *connectors* tracing causality information. In a tagged process of the form $k : P$, the tag k uniquely identifies the process P in a given configuration. We often use the term *key* instead of tag.

The uniqueness of tags in configurations is achieved thanks to the following reduction rule that defines how parallel processes are split.

$$k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

In the above reduction, $\mid$ is the parallel composition operator and ν is the restriction operator, both standard from the π-calculus. As usual, the scope of restriction extends as far to the right as possible. Connector $k \prec (k_1, k_2)$ is used to remember that the process tagged by k has been split into two sub-processes identified by the new keys k_1 and k_2. Thus complex processes can be split into *threads*, where a thread is either a *message*, of the form $a\langle P\rangle$ (where a is a channel name), a receiver process (also called a *trigger*), of the form $a(X) \triangleright_\gamma P$, or a *rollback* instruction of the form roll k, where k is a key.

A *forward* communication step occurs when a message on a channel can be received by a trigger on the same channel. It takes the following form (roll-π is

[5] The version of roll-π presented here is slightly refined w.r.t. the one in [17].

an asynchronous higher-order calculus).

$$(k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q) \longrightarrow \nu k.\, k : Q\{^{P,k}/_{X,\gamma}\} \mid [\mu; k]$$

In this forward step, keys k_1 and k_2 identify threads consisting respectively of a message $a\langle P\rangle$ on channel a and a trigger $a(X) \triangleright_\gamma Q$ expecting a message on channel a. The result of the message input yields, as in higher-order π, the body of the trigger Q with the formal parameter X instantiated by the received value, i.e., process P. Message input also has three side effects: (i) the tagging of the newly created process $Q\{^{P,k}/_{X,\gamma}\}$ by a fresh key k; (ii) the creation of a memory $[\mu; k]$, which records the original two threads,[6] $\mu = (k_1 : a\langle P\rangle) \mid (k_2 : a(X) \triangleright_\gamma Q)$, together with key k; and (iii) the instantiation of variable γ with the newly created key k (the trigger construct is a binder both for its process parameter and its key parameter).

In roll-π, a forward computation, i.e., a series of forward reduction steps as above, can be perfectly undone by backward reductions triggered by the occurrence of an instruction of the form $\mathsf{roll}\ k$, where k refers to a previously instantiated memory. In roll-π, we have for instance the following forward and backward steps, where $M = (k_1 : a\langle Q\rangle) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$:

$$M \longrightarrow \nu k.\, (k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow$$
$$\nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow M$$

The communication between threads k_1 and k_2 in the first step and the split of process k into k_3 and k_4 are perfectly undone by the third (backward) step.

More generally, the set of memories and connectors of a configuration M provides us with an ordering $<:$ between the keys of M that reflects their causal dependency: $k' <: k$ means that key k' has key k as *causal descendant*. Thus, the effects of a rollback can be characterized as follows. When a rollback takes place in a configuration M, triggered by an instruction $k_r : \mathsf{roll}\ k$, it suppresses all threads and processes whose tag is a causal descendant of k, as well as all connectors $k' \prec (k_1, k_2)$ and memories $m = [k_1 : \tau_1 \mid k_2 : \tau_2; k']$ whose key k' is a causal descendant of k. When suppressing such a memory m, the rollback operation may release a thread $k_i : \tau_i$ if k_i is not a causal descendant of k (at least one of the threads of m must have k as causal antecedent if k' has k as causal antecedent). This is due to the fact that a thread that is not a causal descendant of k may be involved in a communication (and then captured into a memory) by a descendant of k. This thread can be seen as a resource that is taken from the environment through interaction, and it should be restored in case of rollback. Finally, rolling-back also releases the content μ of the memory $[\mu; k]$ targeted by the roll, reversing the corresponding communication step.

Flexible reversibility in croll-π. In roll-π, a rollback perfectly undoes a computation originated by a specific message receipt. However, nothing prevents the same

[6] Work can be done to store memories in a more efficient way. We will not consider this issue in the current paper; an approach can be found in [20].

computation from taking place again and again (although not necessarily in the same context, as independent computations may have proceeded on their own in parallel). To allow for flexible reversibility, we extend roll-π with a single new construct, called a *message with alternative*. In croll-π, a message may now take the form $a\langle P\rangle \div C$, where alternative C may either be a message $c\langle Q\rangle \div \mathbf{0}$ with null alternative or the null process $\mathbf{0}$. When the message receipt of $k : a\langle P\rangle \div C$ is rolled-back, configuration $k : C$ is released instead of the original $k : a\langle P\rangle$, as would be the case in roll-π. (Only the alternative associated to the message in the memory $[\mu; k]$ targeted by the roll is released: other processes may be restored, but not modified.) For example, if $M = (k_1 : a\langle Q\rangle \div \mathbf{0}) \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)$ then we have the following computation, where the communication leading to the rollback becomes disabled.

$$
\begin{aligned}
&M \longrightarrow \nu k.\, (k : Q \mid \mathsf{roll}\ k) \mid [M; k] \longrightarrow \\
&\nu k\, k_3\, k_4.\, k \prec (k_3, k_4) \mid k_3 : Q \mid k_4 : \mathsf{roll}\ k \mid [M; k] \longrightarrow \\
&k_1 : \mathbf{0} \mid (k_2 : a(X) \triangleright_\gamma X \mid \mathsf{roll}\ \gamma)
\end{aligned}
$$

We will show that croll-π is powerful enough to devise various kinds of alternatives (see Section 4), whose implementation is not possible in roll-π (cf. Theorem 2). Also, thanks to the higher-order aspect of the calculus, the behavior of roll-π can still be programmed: rigid reversibility can be seen as a particular case of flexible reversibility. Thus, the introduction of messages with alternative has limited impact on the definition of the syntax and of the operational semantics, but it has a strong impact on what can actually be modeled in the calculus and on its theory.

3 The croll-π Calculus: Syntax and Semantics

3.1 Syntax

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 *key variables*, and the set $\mathcal{V}_\mathcal{P}$ of *process variables*. $\mathbb{N}$ denotes the set of natural numbers. 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 $\mathcal{N} \cup \mathcal{K}$; γ range over $\mathcal{V}_\mathcal{K}$; X, Y, Z range over $\mathcal{V}_\mathcal{P}$. We denote by $\tilde{u}$ a finite set $u_1 \ldots u_n$.

Syntax. The syntax of the croll-π calculus is given in Figure 1. *Processes*, given by the P, Q productions, are the standard processes of the asynchronous higher-order π-calculus [25], except for the presence of the roll primitive, the extra bound tag variable in triggers, and messages with alternative that replace roll-π messages $a\langle P\rangle$. The alternative operator $\div$ binds more strongly than any other operator. *Configurations* in croll-π are given by the M, N productions. A configuration is built up from *tagged processes* $k : P$, *memories* $[\mu; k]$, and *connectors* $k \prec (k_1, k_2)$. In a memory $[\mu; k]$, we call μ the *configuration part* of the memory

$$
\begin{array}{rl}
P, Q ::= & \mathbf{0} \mid X \mid \nu a.\, P \mid (P \mid Q) \mid a(X) \triangleright_\gamma P \mid a\langle P \rangle \div C \mid \mathsf{roll}\ k \mid \mathsf{roll}\ \gamma \\
M, N ::= & \mathbf{0} \mid \nu u.\, M \mid (M \mid N) \mid k : P \mid [\mu; k] \mid k \prec (k_1, k_2) \qquad C ::= a\langle P \rangle \div \mathbf{0} \mid \mathbf{0} \\
\mu ::= & (k_1 : a\langle P \rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q) \\
& a, b, c \in \mathcal{N} \quad X, Y, Z \in \mathcal{V}_\mathcal{P} \quad \gamma \in \mathcal{V}_\mathcal{K} \quad u, v, w \in \mathcal{N} \cup \mathcal{K} \quad h, k, l \in \mathcal{K}
\end{array}
$$

Fig. 1. Syntax of croll-π

and k its *key*. $\mathcal{P}$ denotes the set of croll-π processes and $\mathcal{C}$ the set of croll-π configurations. We let (together with their decorated variants) P, Q, R range over $\mathcal{P}$ and L, M, N range over $\mathcal{C}$. We call *thread* a process that is either a message with alternative $a\langle P \rangle \div C$, a trigger $a(X) \triangleright_\gamma P$, or a rollback instruction $\mathsf{roll}\ k$. We let τ and its decorated variants range over threads. We write $\prod_{i \in I} M_i$ for the parallel composition of configurations M_i for each $i \in I$ (by convention $\prod_{i \in I} M_i = \mathbf{0}$ if $I = \emptyset$), and we abbreviate $a\langle \mathbf{0} \rangle$ to $\overline{a}$.

Free identifiers and free variables. Notions of free identifiers and free variables in croll-π are standard. 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 key variable γ with scope P. We denote by $\mathtt{fn}(P)$ and $\mathtt{fn}(M)$ the set of free names and keys of process P and configuration M, respectively. Note in particular that $\mathtt{fn}(k : P) = \{k\} \cup \mathtt{fn}(P)$, $\mathtt{fn}(\mathsf{roll}\ k) = \{k\}$. We say that a process P or a configuration M is *closed* if it has no free (process or key) variable. We denote by $\mathcal{P}_{cl}$ and $\mathcal{C}_{cl}$ the sets of closed processes and configurations, respectively. We abbreviate $a(X) \triangleright_\gamma P$, where X is not free in P, to $a \triangleright_\gamma P$; and $a(X) \triangleright_\gamma P$, where γ is not free in P, to $a(X) \triangleright P$.

Remark 1. We have no construct for replicated processes or internal choice in croll-π: as in the higher-order π-calculus, these can easily be encoded.

Remark 2. In the remainder of the paper, 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.

3.2 Reduction Semantics

The reduction semantics of croll-π is defined via a reduction relation $\longrightarrow$, which is a binary relation over closed configurations ($\longrightarrow \subset \mathcal{C}_{cl} \times \mathcal{C}_{cl}$), and a structural congruence relation $\equiv$, which is a binary relation over configurations ($\equiv \subset \mathcal{C} \times \mathcal{C}$). We define *configuration contexts* as "configurations with a hole $\bullet$", given by the grammar: $\mathbb{C} ::= \bullet \mid (M \mid \mathbb{C}) \mid \nu u.\, \mathbb{C}$. *General contexts* $\mathbb{G}$ are just configurations with a hole $\bullet$ in a place where an arbitrary process P can occur. A *congruence* on processes or configurations is an equivalence relation $\mathcal{R}$ that is closed for general or configuration contexts: $P\, \mathcal{R}\, Q \implies \mathbb{G}[P]\, \mathcal{R}\, \mathbb{G}[Q]$ and $M\, \mathcal{R}\, N \implies \mathbb{C}[M]\, \mathcal{R}\, \mathbb{C}[N]$.

$$\text{(E.ParC)}\ M \mid N \equiv N \mid M \qquad \text{(E.ParA)}\ M_1 \mid (M_2 \mid M_3) \equiv (M_1 \mid M_2) \mid M_3$$

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

$$\text{(E.NewC)}\ \nu u.\, \nu v.\, M \equiv \nu v.\, \nu u.\, M \qquad \text{(E.NewP)}\ (\nu u.\, M) \mid N \equiv \nu u.\, (M \mid N)$$

$$\text{(E.}\alpha\text{)}\ M =_\alpha N \implies M \equiv N \qquad \text{(E.TagC)}\ k \prec (k_1, k_2) \equiv k \prec (k_2, k_1)$$

$$\text{(E.TagA)}\ \nu h.\, k \prec (h, k_3) \mid h \prec (k_1, k_2) \equiv \nu h.\, k \prec (k_1, h) \mid h \prec (k_2, k_3)$$

Fig. 2. Structural congruence for croll-π

$$\text{(S.Com)}\ \frac{\mu = (k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2)}{(k_1 : a\langle P\rangle \div C) \mid (k_2 : a(X) \triangleright_\gamma Q_2) \longrightarrow \nu k.\, (k : Q_2\{^{P,k}/_{X,\gamma}\}) \mid [\mu; k]}$$

$$\text{(S.TagN)}\ k : \nu a.\, P \longrightarrow \nu a.\, k : P$$

$$\text{(S.TagP)}\ k : P \mid Q \longrightarrow \nu k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : P \mid k_2 : Q$$

$$\text{(S.Roll)}\ \frac{k <: N \qquad \texttt{complete}(N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k)) \qquad \mu' = \texttt{xtr}(\mu)}{N \mid [\mu; k] \mid (k_r : \mathsf{roll}\ k) \longrightarrow \mu' \mid N\not\downarrow_k}$$

$$\text{(S.Ctx)}\ \frac{M \longrightarrow N}{\mathbb{C}[M] \longrightarrow \mathbb{C}[N]} \qquad \text{(S.Eqv)}\ \frac{M \equiv M' \qquad M' \longrightarrow N' \qquad N' \equiv N}{M \longrightarrow N}$$

Fig. 3. Reduction rules for croll-π

Structural congruence $\equiv$ is defined as the smallest congruence on configurations that satisfies the axioms in Figure 2, where $t =_\alpha t'$ denotes equality of t and t' modulo α-conversion. Axioms E.ParC to E.α are standard from the π-calculus. Axioms E.TagC and E.TagA model commutativity and associativity of connectors, in order not to have a rigid tree structure. Thanks to axiom E.NewC, $\nu\tilde{u}.\, A$ stands for $\nu u_1 \ldots u_n.\, A$ if $\tilde{u} = u_1\ \ldots\ u_n$.

Configurations can be written in normal form using structural congruence.

Lemma 1 (Normal form). *Given a configuration M, we have:*

$$M \equiv \nu\tilde{n}.\, \prod_i (k_i : P_i) \mid \prod_j [\mu_i; k_j] \mid \prod_l k_l \prec (k'_l, k''_l)$$

The reduction relation $\longrightarrow$ is defined as the smallest binary relation on closed configurations satisfying the rules of Figure 3. This extends the naïve semantics of

roll-π introduced in [17],[7] and outlined here in Section 2, to manage alternatives. We denote by $\Longrightarrow$ the reflexive and transitive closure of $\longrightarrow$.

Reductions are either forward, given by rules S.Com, S.TagN, and S.TagP, or backward, defined by rule S.Roll. They are closed under configuration contexts (rule S.Ctx) and under structural congruence (rule S.Eqv). The rule for communication S.Com is the standard communication rule of the higher-order π-calculus with the side effects discussed in Section 2. Rule S.TagN allows restrictions in processes to be lifted at the configuration level. Rule S.TagP allows to split parallel processes. Rule S.Roll enacts rollback, canceling all the effects of the interaction identified by the unique key k, and releasing the initial configuration that gave rise to the interaction, where the alternative replaces the original message. This is the only difference between croll-π and roll-π: in the latter, the memory μ was directly released. However, this small modification yields significant changes to the expressive power of the calculus, as we will see later.

The rollback impacts only the causal descendants of k, defined as follows.

Definition 1 (Causal dependence). *Let M be a configuration and let $\mathcal{T}_M$ be the set of keys occurring in M. Causal dependence $<:_M$ is the reflexive and transitive closure of $<_M$, which is defined as the smallest binary relation on $\mathcal{T}_M$ satisfying the following clauses:*

- *$k <_M k'$ if $k \prec (k_1, k_2)$ occurs in M with $k' = k_1$ or $k' = k_2$;*
- *$k <_M k'$ if a thread $k : P$ occurs (inside μ) in a memory $[\mu; k']$ of M.*

If the configuration M is clear from the context, we write $k <: k'$ for $k <:_M k'$.

A backward reduction triggered by roll k involves *all* and *only* the descendants of key k. We ensure they are all selected by requiring that the configuration is *complete*, and that no other term is selected by requiring *k-dependence*.

Definition 2 (Complete configuration). *A configuration M is* complete*, denoted as* $\texttt{complete}(M)$*, if, for each memory $[\mu; k]$ and each connector $k' \prec (k, k_1)$ or $k' \prec (k_1, k)$ that occurs in M there exists in M either a connector $k \prec (h_1, h_2)$ or a tagged process $k : P$ (possibly inside a memory).*

A configuration M is k-dependent if all its components depend on k.

Definition 3 (k-dependence). *Let M be a configuration such that: $M \equiv \nu\tilde{u}.\ \prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[\mu_j; k_j] \mid \prod_{l\in L} k_l \prec (k'_l, k''_l)$ with $k \notin \tilde{u}$. Configuration M is k-dependent, written $k <: M$ by overloading the notation for causal dependence among keys, if for every i in $I \cup J \cup L$, we have $k <:_M k_i$.*

Rollback should release all the resources consumed by the computation to be rolled-back which were provided by other threads. They are computed as follows.

[7] We extend the naïve semantics instead of the high-level or the low-level semantics (also defined in [17]) for the sake of simplicity. However, reduction semantics corresponding to the high-level and low-level semantics of roll-π can similarly be specified.

Definition 4 (Projection). *Let M be a configuration such that:* $M \equiv \nu\tilde{u}.\, \prod_{i\in I}(k_i : P_i) \mid \prod_{j\in J}[k'_j : R_j \mid k''_j : T_j; k_j] \mid \prod_{l\in L} k_l \prec (k'_l, k''_l)$ *with* $k \notin \tilde{u}$. *Then:*

$$M \not\downarrow_k = \nu\tilde{u}.\, \Big(\prod_{j'\in J'} k'_{j'} : R_{j'}\Big) \mid \Big(\prod_{j''\in J''} k''_{j''} : T_{j''}\Big)$$

where $J' = \{j \in J \mid k \not<: k'_j\}$ *and* $J'' = \{j \in J \mid k \not<: k''_j\}$.

Intuitively, $M \not\downarrow_k$ consists of the threads inside memories in M which are not dependent on k.

Finally, and this is the main novelty of croll-π, function `xtr` defined below replaces messages from the memory targeted by the roll by their alternatives.

Definition 5 (Extraction function).

$$\texttt{xtr}(M \mid N) = \texttt{xtr}(M) \mid \texttt{xtr}(N) \qquad \texttt{xtr}(k : a\langle P\rangle \div C) = k : C$$
$$\texttt{xtr}(k : a(X) \triangleright_\gamma Q) = k : a(X) \triangleright_\gamma Q$$

No other case needs to be taken into account as `xtr` is only called on the contents of memories.

Remark 3. Not all syntactically licit configurations make sense. In particular, we expect configurations to respect the causal information required for executing croll-π programs. We therefore work only with *coherent* configurations. A configuration is coherent if it is obtained by reduction starting from a configuration of the form $\nu k.\, k : P$ where P is closed and contains no roll h primitive (all the roll primitives should be of the form roll γ).

3.3 Barbed Congruence

We define notions of strong and weak barbed congruence to reason about croll-π processes and configurations. Name a is *observable* in configuration M, denoted as $M\downarrow_a$, if $M \equiv \nu\tilde{u}.\, (k : a\langle P\rangle \div C) \mid N$, with $a \notin \tilde{u}$. 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 6 (Barbed congruences for configurations). *A relation* $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *on closed configurations is a* strong *(respectively* weak*)* barbed simulation *if whenever* $M\,\mathcal{R}\,N$,

- $M\downarrow_a$ *implies* $N\downarrow_a$ *(respectively* $N \Longrightarrow\downarrow_a$*);*
- $M \longrightarrow M'$ *implies* $N \longrightarrow N'$ *(respectively* $N \Longrightarrow N'$*) with* $M'\mathcal{R}N'$.

A relation $\mathcal{R} \subseteq \mathcal{C}_{cl} \times \mathcal{C}_{cl}$ *is a* strong (weak) barbed bisimulation *if* $\mathcal{R}$ *and* $\mathcal{R}^{-1}$ *are strong (weak) barbed simulations. We call* strong (weak) barbed bisimilarity *and denote by* $\sim$ $(\approx)$ *the largest strong (weak) barbed bisimulation. The largest congruence for configuration contexts included in* $\sim$ $(\approx)$ *is called* strong (weak) barbed congruence*, denoted by* $\sim_c$ $(\approx_c)$.

The notion of strong and weak barbed congruence extends to closed and open processes, by considering general contexts that form closed configurations.

Definition 7 (Barbed congruences for processes). *A relation $\mathcal{R} \subseteq \mathcal{P}_{cl} \times \mathcal{P}_{cl}$ on closed processes is a* strong *(resp.* weak*) barbed congruence if whenever $P \mathcal{R} Q$, for all general contexts $\mathbb{G}$ such that $\mathbb{G}[P]$ and $\mathbb{G}[Q]$ are closed configurations, we have $\mathbb{G}[P] \sim_c \mathbb{G}[Q]$ (resp. $\mathbb{G}[P] \approx_c \mathbb{G}[Q]$).*

Two open processes P and Q are said to be strong (resp. weak) barbed congruent, denoted by $P \sim_c^o Q$ (resp. $P \approx_c^o Q$) if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, we have $P\sigma \sim_c Q\sigma$ (resp. $P\sigma \approx_c Q\sigma$).

Working with arbitrary contexts can quickly become unwieldy. We offer the following Context Lemma to simplify the proofs of congruence.

Theorem 1 (Context lemma). *Two processes P and Q are weak barbed congruent, $P \approx_c^o Q$, if and only if for all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, all closed configurations M, and all keys k, we have: $M \mid (k : P\sigma) \approx M \mid (k : Q\sigma)$.*

The proof of this Context Lemma is much more involved than the corresponding one in the π-calculus, notably because of the bookkeeping required in dealing with process and thread tags. It is obtained by composing the lemmas below.

The first lemma shows that the only relevant configuration contexts are parallel contexts.

Lemma 2 (Context lemma for closed configurations). *For any closed configurations M, N, $M \sim_c N$ if and only if, for all closed configurations L, $M \mid L \sim N \mid L$. Likewise, $M \approx_c N$ if and only if, for all L, $M \mid L \approx N \mid L$.*

Proof. The left to right implication is immediate, by definition of $\sim_c$. For the other direction, the proof consists in showing that $\mathcal{R} = \{\langle \mathbb{C}[M], \mathbb{C}[N] \rangle \mid \forall L, M \mid L \sim N \mid L\}$ is included in $\sim$. The weak case is identical to the strong one. □

We can then prove the thesis on closed processes.

Lemma 3 (Context lemma for closed processes). *Let P and Q be closed processes. We have $P \approx_c Q$ if and only if, for all closed configuration contexts $\mathbb{C}$ and $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P] \approx \mathbb{C}[k : Q]$.*

Proof. The left to right implication is clear. One can prove the right to left direction by induction on the form of general contexts for processes, using Lemma 4 below for message contexts. □

Lemma 4 (Factoring). *For all closed processes P, all closed configurations M such that $M\{^P/_X\}$ is closed, and all $c, t, k_0, k_0' \notin \mathtt{fn}(M, P)$, we have*

$$M\{^P/_X\} \approx_c \nu c, t, k_0, k_0'.\, M\{^{\overline{c}}/_X\} \mid k_0 : t\langle Y_P \rangle \mid k_0' : Y_P$$

where $Y_P = t(Y) \triangleright (c \triangleright P) \mid t\langle Y \rangle \mid Y$.

We then deal with open processes.

Lemma 5 (Context lemma for open processes). *Let P and Q be (possibly open) processes. We have $P \approx_c^o Q$ if and only if for all closed configuration contexts $\mathbb{C}$, all substitutions σ such that $P\sigma$ and $Q\sigma$ are closed, and all $k \notin \mathtt{fn}(P, Q)$, we have $\mathbb{C}[k : P\sigma] \approx \mathbb{C}[k : Q\sigma]$.*

Proof. For the only if part, one proceeds by induction on the number of bindings in σ. The case for zero bindings follows from Lemma 3. For the inductive case, we write $\mathbb{P}[\bullet]$ for a process where an occurrence of $\mathbf{0}$ has been replaced by $\bullet$, and we show that contexts of the form $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright \mathbb{P}'[\bullet]$ where a is fresh and $\mathbb{P} = a\langle R\rangle \mid a(X) \triangleright_\gamma \mathbb{P}'[\bullet]$ where a is fresh and X never occurs in the continuation actually enforce the desired binding.

For the if part, the proof is by induction on the number of triggers. If the number of triggers is 0 then the thesis follows from Lemma 3. The inductive case consists in showing that equivalence under substitutions ensures equivalence under a trigger context. □

Proof (of Theorem 1). A direct consequence of Lemma 5 and Lemma 2. □

4 croll-π Expressiveness

4.1 Alternative Idioms

The message with alternative $a\langle P\rangle \div C$ triggers alternative C upon rollback. We choose to restrict C to be either a message with $\mathbf{0}$ alternative or $\mathbf{0}$ itself in order to have a minimal extension of roll-π. However, this simple form of alternative is enough to encode far more complex alternative policies and constructs, as shown below. We define the semantics of the alternative idioms below by only changing function `xtr` in Definition 5. We then encode them in croll-π and prove the encoding correct w.r.t. weak barbed congruence. More precisely, for every extension below the notion of barbs is unchanged. The notion of barbed bisimulation thus relates processes with slightly different semantics (only `xtr` differs) but sharing the same notion of barbs. Since we consider extensions of croll-π, in weak barbed congruence we consider just closure under croll-π contexts. By showing that the extensions have the same expressive power of croll-π, we ensure that allowing them in contexts would not change the result. Every encoding maps unmentioned constructs homomorphically to themselves. After having defined each alternative idiom, we freely use it as an abbreviation.

Arbitrary alternatives. Messages with arbitrary alternative can be defined by allowing C to be any process Q. No changes are required to the definition of function `xtr`. We can encode arbitrary alternatives as follows, where c is not free in P, Q.

$$(\!|a\langle P\rangle \div Q|\!)_{aa} = \nu c.\, a\langle (\!|P|\!)_{aa}\rangle \div c\langle (\!|Q|\!)_{aa}\rangle \div \mathbf{0} \mid c(X) \triangleright X$$

Proposition 1. *$P \approx_c (\!|P|\!)_{aa}$ for any closed process with arbitrary alternatives.*

$$\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \mathcal{R}_4 \cup \mathcal{R}_5 \cup Id$$
$$\mathcal{R}_1 = \{\langle k : a\langle P\rangle \div Q \mid L\ ,\ k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L\rangle\}$$
$$\mathcal{R}_2 = \{\langle k : a\langle P\rangle \div Q \mid L\ ,\ \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L\rangle\}$$
$$\mathcal{R}_3 = \{\langle \nu h.\, [k : a\langle P\rangle \div Q \mid k' : a(X) \triangleright_\gamma R; h] \mid L''\ ,$$
$$\nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid k' : a(X) \triangleright_\gamma R; h] \mid k_2 : c(X) \triangleright X \mid L''\rangle\}$$
$$\mathcal{R}_4 = \{\langle k : Q \mid L'''\ ,\ \nu c\, k_1\, k_2.\, k \prec (k_1, k_2) \mid k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X \mid L'''\rangle\}$$
$$\mathcal{R}_5 = \{\langle k : Q \mid L'''\ ,\ \nu c\, k_1\, k_2\, h.\, k \prec (k_1, k_2) \mid [k_1 : c\langle Q\rangle \div \mathbf{0} \mid k_2 : c(X) \triangleright X; h] \mid h : Q \mid L'''\rangle\}$$

Fig. 4. Bisimulation relation for arbitrary alternatives

Proof. We consider just one instance of arbitrary alternative, the thesis will follow by transitivity.

Thanks to Lemma 5 and Lemma 2, we only need to prove that for all closed configurations L and $k \notin \mathtt{fn}(P)$, we have $k : a\langle P\rangle \div Q \mid L \approx k : (\nu c.\, a\langle P\rangle \div c\langle Q\rangle \div \mathbf{0} \mid c(X) \triangleright X) \mid L$. We consider the relation $\mathcal{R}$ in Figure 4 and prove that it is a weak barbed bisimulation. In every relation, L is closed and $k \notin \mathtt{fn}(P)$.

In $\mathcal{R}_1$, the right configuration can reduce via rule S.TagN followed by S.TagP. These lead to $\mathcal{R}_2$. Performing these reductions is needed to match the barb and the relevant reductions of the left configuration, thus we consider directly $\mathcal{R}_2$. In $\mathcal{R}_2$ the barbs coincide. Rollbacks lead to the identity. The only possible communication is on a, and requires $L \equiv L' \mid k' : a(X) \triangleright_\gamma R$. It leads to $\mathcal{R}_3$, where $L'' = L' \mid R\{^{P,h}/_{X,\gamma}\}$. In $\mathcal{R}_3$ the barbs coincide too. All the reductions can be matched by staying in $\mathcal{R}_3$ or going to the identity, but for executing a roll with key h. This leads to $\mathcal{R}_4$, where L''' is closed. From $\mathcal{R}_4$ we can always execute the internal communication at c leading to $\mathcal{R}_5$. The thesis follows from the result below, whose proof requires again to find a suitable bisimulation relation.

Lemma 6. *For each configuration M k-dependent and complete such that k', t, $k_1, k_2 \notin \mathtt{fn}(M)$ we have $M \approx_c \nu k'\, t\, k_1\, k_2.\, k \prec (k_1, k_2) \mid [k_1 : t\langle Q\rangle \div C \mid k_2 : t(X) \triangleright R; k'] \mid M\{^{k'}/_{k}\}$.*

□

Proofs concerning other idioms follow similar lines, and can be found in the online technical report [16].

A particular case of arbitrary alternative $a\langle P\rangle \div Q$ is when Q is a message whose alternative is not $\mathbf{0}$. By applying this pattern recursively we can write $a_1\langle P_1\rangle \div \ldots \div a_n\langle P_n\rangle \div Q$. In particular, by choosing $a_1 = \cdots = a_n$ and $P_1 = \cdots = P_n$ we can try n times the alternative P before giving up by executing Q.

Endless retry. We can also retry the same alternative infinitely many times, thus obtaining the behavior of roll-π messages. These messages can be integrated into croll-π semantics by defining function `xtr` as the identity on them.

$$(\!|a\langle P\rangle|\!)_{er} = \nu t.\, Y \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Y\rangle \qquad Y = t(Z) \triangleright Z \mid a\langle (\!|P|\!)_{er}\rangle \div t\langle Z\rangle$$

Proposition 2. *$P \approx_c (\!|P|\!)_{er}$ for any closed process with roll-π messages.*

As corollary of Proposition 2 we thus have the following.

Corollary 1. *croll-π is a conservative extension of roll-π.*

Triggers with alternative. Until now we attached alternatives to messages. Symmetrically, one may attach alternatives to triggers. Thus, upon rollback, the message is released and the trigger is replaced by a new process.

The syntax for triggers with alternative is $(a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}$. As for messages, we use a single message as alternative, but one can use general processes as described earlier. Triggers with alternative are defined by the extract clause below.

$$\mathtt{xtr}(k : (a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}) = k : b\langle Q' \rangle \div \mathbf{0}$$

Interestingly, messages with alternative and triggers with alternative may coexist. The encoding of triggers with alternative is as follows.

$$(\!|(a(X) \triangleright_\gamma Q) \div b\langle Q' \rangle \div \mathbf{0}|\!)_{at} = \nu c\, d.\, \overline{c} \div \overline{d} \div \mathbf{0} \mid (c \triangleright_\gamma a(X) \triangleright (\!|Q|\!)_{at}) \mid (d \triangleright b\langle (\!|Q'|\!)_{at} \rangle \div \mathbf{0})$$

Proposition 3. *$P \approx_c (\!|P|\!)_{at}$ for any closed process with triggers with alternative.*

4.2 Comparing croll-π and roll-π

While Corollary 1 shows that croll-π is at least as expressive as roll-π, a natural question is whether croll-π is actually strictly more expressive than roll-π or not. The theorem below gives a positive answer to this question.

Theorem 2. *There is no encoding $(\!|\bullet|\!)$ from croll-π to roll-π such that for each croll-π configuration M:*

1. *if M has a computation including at least a backward step, then $(\!|M|\!)$ has a computation including at least a backward step;*
2. *if M has only finite computations, then $(\!|M|\!)$ has only finite computations.*

Proof. Consider configuration $M = \nu k.\, k : \overline{a} \div \overline{b} \div \mathbf{0} \mid a \triangleright_\gamma \mathsf{roll}\ \gamma$. This configuration has a unique possible computation, composed by one forward step followed by one backward step. Assume towards a contradiction that an encoding exists and consider $(\!|M|\!)$. $(\!|M|\!)$ should have at least a computation including a backward step. From roll-π loop lemma [17, Theorem 1], if we have a backward step, we are able to go forward again, and then there is a looping computation. This is in contrast with the second condition of the encoding. The thesis follows. □

The main point behind this result is that the Loop Lemma, a cornerstone of roll-π theory [17] capturing the essence of rigid rollback (and similar results in [8, 18, 20, 22]), does not hold in croll-π. Naturally, the result above does not imply that croll-π cannot be encoded in HOπ or in π-calculus. However, these calculi are too low level for us, as hinted at by the fact that the encoding of a simple reversible higher order calculus into HOπ is quite complex, as shown in [18].

$$
\begin{aligned}
Q_i \;\triangleq\; & (act_i(Z) \triangleright p_i\langle i,1\rangle \div \ldots \div p_i\langle i,8\rangle \div f_i\langle \mathbf{0}\rangle \div \mathbf{0} \mid \\
& (p_i(\mathbf{x_i}) \triangleright_{\gamma_i} !c_i\langle \mathbf{x_i}\rangle \div \mathbf{0} \mid act_{i+1}\langle \mathbf{0}\rangle \mid f_{i+1}(Y) \triangleright \mathsf{roll}\ \gamma_i \mid \\
& \textstyle\prod_{j=1}^{i-1} c_j(\mathbf{y_j}) \triangleright \mathtt{if}\ err(\mathbf{x_i},\mathbf{y_j})\ \mathtt{then}\ \mathsf{roll}\ \gamma_i)) \\
err((x_1,x_2),(y_1,y_2)) \;\triangleq\; & (x_1 = y_1 \vee x_2 = y_2 \vee |x_1 - y_1| = |x_2 - y_2|)
\end{aligned}
$$

Fig. 5. The i-th queen

5 Programming in croll-π

A main goal of croll-π is to make reversibility techniques exploitable for application development. Even if croll-π is not yet a full-fledged language, we have developed a proof-of-concept interpreter for it. To the best of our knowledge, this is the first interpreter for a causal-consistent reversible language. We then put the interpreter at work on a few simple, yet interesting, programming problems. We detail below the algorithm we devised to solve the Eight Queens problem [3, p. 165]. The interpreter and the code for solving the Eight Queens problem are available at `http://proton.inrialpes.fr/~mlienhar/croll-pi/implem`, together with examples of encodings of primitives for error handling, and an implementation of the car repair scenario of the EU project Sensoria.

The interpreter for croll-π is written in Maude [10], a language based on both equational and rewriting logic that allows the programmer to define terms and reduction rules, e.g., to execute reduction semantics of process calculi. Most of croll-π's rules are straightforwardly interpreted, with the exception of rule S.Roll. This rule is quite complex as it involves checks on an unbounded number of interacting components. Such an issue is already present in roll-π [17], where it is addressed by providing an easier to implement, yet equivalent, low-level semantics. This semantics replaces rule S.Roll with a protocol that sends notifications to all the involved components to roll-back, then waits for them to do so. Extending the low-level semantics from roll-π to croll-π simply requires the application of function `xtr` to the memory targeted by the rollback. We do not detail the low-level semantics of croll-π here, and refer the reader to [17] for a detailed description in the setting of roll-π. Our Maude interpreter is based on this low-level semantics, extended with values (integers and pairs) and with the `if-then-else` construct. It is fairly concise (less than 350 lines of code).

The Eight Queens problem is a well-known constraint-programming problem which can be formulated as follows: how to place 8 queens on an 8×8 chess board so that no queen can directly capture another? We defined an algorithm in croll-π where queens are concurrent entities, numbered from 1 to 8, all executing the code schema shown in Figure 5. We use $\mathbf{x}$ to indicate a pair of integer variables (x_1, x_2), and replicated messages $!c_i\langle \mathbf{x}\rangle \div \mathbf{0}$ to denote the encoding of a parallel composition of an infinite number of messages $c_i\langle \mathbf{x}\rangle \div \mathbf{0}$ (cf. Remark 1).

The queens are activated in numeric order. The i-th queen is activated by a message on channel act_i from its predecessor (a message on act_1 is needed to start the whole computation). When a queen is activated it looks for its position by trying sequentially all the positions in the i-th row of the chess board. To try a position, it sends it over channel p_i. Then, the position is made available on channel c_i and the next queen is activated. Finally, the position is checked for compatibility with the positions of previous queens. This is done by computing (in parallel) $err(\mathbf{x_j}, \mathbf{x_i})$ for each $j < i$. If a check fails, $\mathsf{roll}\ \gamma_i$ rolls-back the choice of the position of queen i. The alternatives mechanism allows to try the next position. If no suitable position is available, the choice of position of queen $i-1$ is rolled-back (possibly recursively) by the communication over f_i. Note that a roll-back of queen j makes all queens i with $i > j$ restart, since previously discarded positions may now be acceptable. This is obtained thanks to activation messages establishing the needed causal dependencies. When the computation ends, messages on c_i contain positions which are all compatible.

6 Asynchronous Interacting Transactions

This section shows how croll-π can model in a precise way interacting transactions with compensations as formalized in TransCCS [13]. Actually, the natural croll-π encoding improves on the semantics in [13], since croll-π causality tracking is more precise than the one in TransCCS, which is based on dynamic embedding of processes into transactions. Thus croll-π avoids some spurious undo of actions, as described below. Before entering the details of TransCCS, let us describe the general idea of transaction encoding.

We consider a very general notion of atomic (but not necessarily isolated) transaction, i.e., a process that executes completely or not at all. Informally, a transaction $[P, Q]_\gamma$ with name γ executing process P with compensation Q can be modeled by a process of the form:

$$[P, Q]_\gamma = \nu a\, c.\, \overline{a} \div \overline{c} \div \mathbf{0} \mid (a \triangleright_\gamma P) \mid (c \triangleright Q)$$

Intuitively, when $[P, Q]_\gamma$ is executed, it first starts process P under the rollback scope γ. Abortion of the transaction can be triggered in P by executing a $\mathsf{roll}\ \gamma$. Whenever P is rolled-back, the rollback does not restart P (since the message on a is substituted by the alternative on c), but instead starts the compensation process Q. In this approach commit is implicit: when there is no reachable $\mathsf{roll}\ \gamma$, the transaction is committed. From the explanation above, it should be clear that in the execution of $[P, Q]_\gamma$, either P executes completely, i.e., until it reaches a commit, or not at all, in the sense that it is perfectly rolled-back. If P is ever rolled-back, its failed execution can be compensated by that of process Q. Interestingly, and in contrast with irreversible actions used in [12], our rollback scopes can be nested without compromising this all-or-nothing semantics.

Let us now consider an asynchronous fragment of TransCCS [13], removing choice and recursion. Dealing with the whole calculus would not add new difficulties related to rollback, but only related to the encoding of such operators in

$$\text{(R-Comm)}\quad \overline{a} \mid a.P \longrightarrow P \qquad \text{(R-Emb)}\quad \frac{k \notin \mathtt{fn}(R)}{[\![P \triangleright_k Q]\!] \mid R \longrightarrow [\![P \mid R \triangleright_k Q \mid R]\!]}$$

$$\text{(R-Co)}\quad [\![P \mid \mathsf{co}\ k \triangleright_k Q]\!] \longrightarrow P \qquad \text{(R-Ab)}\quad [\![P \triangleright_k Q]\!] \longrightarrow Q$$

and is closed under active contexts $\nu a.\bullet$, $\bullet \mid Q$ and $[\![\bullet \triangleright_k Q]\!]$, and structural congruence.

Fig. 6. Reduction rules for TransCCS

higher-order π. The syntax of the fragment of TransCCS we consider is:

$$P ::= \mathbf{0} \mid \nu a.\, P \mid (P \mid Q) \mid \overline{a} \mid a.P \mid \mathsf{co}\ k \mid [\![P \triangleright_k Q]\!]$$

Essentially, it extends CCS with a transactional construct $[\![P \triangleright_k Q]\!]$, executing a transaction with body P, name k and compensation Q, and a commit operator $\mathsf{co}\ k$.

The rules defining the semantics of TransCCS are given in Figure 6. Structural congruence contains the usual rules for parallel composition and restriction. Keep in mind that transaction scope is a binder for its name k, thus k does not occur outside the transaction, and there is no name capture in rules R-Co and R-Emb.

A croll-π transaction $[P, Q]_\gamma$ as above has explicit abort, specified by $\mathsf{roll}\ \gamma$, where γ is used as the transaction name, and implicit commit. TransCCS takes different design choices, using non-deterministic abort and programmable commit. Thus we have to instantiate the encoding above.

Definition 8 (TransCCS encoding). *Let P be a TransCCS process. Its encoding $(\!|\bullet|\!)_t$ in croll-π is defined as:*

$$(\!|\nu a.\, P|\!)_t = \nu a.\, (\!|P|\!)_t \qquad (\!|P \mid Q|\!)_t = (\!|P|\!)_t \mid (\!|Q|\!)_t \qquad (\!|\overline{a}|\!)_t = \overline{a}$$
$$(\!|a.P|\!)_t = a \triangleright (\!|P|\!)_t \qquad (\!|\mathsf{co}\ l|\!)_t = l(X) \triangleright \mathbf{0} \qquad (\!|\mathbf{0}|\!)_t = \mathbf{0}$$

$$(\!|[\![P \triangleright_l Q]\!]|\!)_t = [\nu l.\, (\!|P|\!)_t \mid l\langle \mathsf{roll}\ \gamma\rangle \mid l(X) \triangleright X, (\!|Q|\!)_t]_\gamma$$

Since in croll-π only configurations can execute, the behavior of P should be compared with $\nu k.\, k : (\!|P|\!)_t$.

In the encoding, abort is always possible since at any time the only occurrence of the roll in the transaction can be activated by a communication on l. On the other hand, executing the encoding of a TransCCS commit disables the roll related to the transaction. This allows to garbage collect the compensation, and thus corresponds to an actual commit. Note, however, that in croll-π the abort operation is not atomic as in TransCCS since the roll related to a transaction first has to be enabled through a communication on l, disabling in this way any possibility to commit, and then it can be executed. Clearly, until the roll is executed, the body of the transaction can continue its execution. To make abort atomic one would need the ability to disable an active roll, as could be done

using a (mixed) choice such as $(\mathsf{roll}\ k) + (l \triangleright \mathbf{0})$. In this setting an output on l would commit the transaction. Adding choice would not make the reduction semantics more difficult, but its impact on behavioral equivalence has not been studied yet.

The relation between the behavior of a TransCCS process P and of its translation $(\!|P|\!)_t$ is not immediate, not only because of the comment above on atomicity, but also because of the approximate tracking of causality provided by TransCCS. TransCCS tracks interacting processes using rule (R-Emb): only processes inside the same transaction may interact, and when a process enters the transaction it is saved in the compensation, so that it can be restored in case of abort. However, no check is performed to ensure that the process actually interacts with the transaction code. For instance, a process $\overline{a} \mid a.P$ may enter a transaction $[\![Q \triangleright_k R]\!]$ and then perform the communication at a. Such a communication would be undone in case of abort. This is a spurious undo, since the communication at a is not related to the transaction code. Actually, the same communication could have been performed outside the transaction, and in this case it would not have been undone.

In croll-π encoding, a process is "inside" the transaction with key k if and only if its tag is causally dependent on k. Thus a process enters a transaction only by interacting with a process inside it. For this reason, there is no reduction in croll-π corresponding to rule (R-Emb), and since no process inside the transaction is involved in the reduction at a above, the reduction would not be undone in case of abort, since it actually happens "outside" the transaction. Thus our encoding avoids spurious undo, and computations in croll-π correspond to computations in TransCCS with minimal applications of rule (R-Emb). These computations are however very difficult to characterize because of syntactic constraints. In fact, for two processes inside two parallel transactions k_1 and k_2 to interact, either k_1 should move inside k_2 or vice versa, but in both the cases not only the interacting processes move, as minimality would require, but also all the other processes inside the same transactions have to move. Intuitively, TransCCS approximates the causality relation, which is a dag, using the tree defined by containment. The spurious reductions undone in TransCCS can always be redone so to reach a state corresponding to the croll-π one. In this sense croll-π minimizes the set of interactions undone.

We define a notion of weak barbed bisimilarity ${}_t\!\approx_{c\pi}$ relating a TransCCS process P and a croll-π configuration M. First, we define barbs in TransCCS by the predicate $P{\downarrow_a}$, which is true in the cases below, false otherwise.

$$\begin{array}{ll} \overline{a}{\downarrow_a} & \nu b.\, P{\downarrow_a} \ \text{ if } P{\downarrow_a} \wedge a \neq b \\ P \mid P'{\downarrow_a} \ \text{ if } P{\downarrow_a} \vee P'{\downarrow_a} & [\![P \triangleright_k Q]\!]{\downarrow_a} \ \text{ if } P{\downarrow_a} \wedge a \neq k \end{array}$$

Here, differently from [13], we observe barbs inside the transaction body, to have a natural correspondence with croll-π barbs.

Definition 9. *A relation $\mathcal{R}$ relating TransCCS processes P and croll-π configurations M is a weak barbed bisimulation if and only if for each $(P, M) \in \mathcal{R}$:*

1. *if* $P{\downarrow_a}$ *then* $M \Longrightarrow{\downarrow_a}$;
2. *if* $M{\downarrow_a}$ *then* $P \Longrightarrow{\downarrow_a}$;
3. *if* $P \longrightarrow P_1$ *is derived using rule* (R-Ab) *then* $M \Longrightarrow M'$, $P_1 \Longrightarrow P_2$ *and* $P_2 \mathcal{R} M'$;
4. *if* $P \longrightarrow P_1$ *is derived without using rule* (R-Ab) *then* $M \Longrightarrow M'$ *and* $P_1 \mathcal{R} M'$;
5. *if* $M \longrightarrow M'$ *then either: (i)* $P \mathcal{R} M'$ *or (ii)* $P \longrightarrow P_1$ *and* $P_1 \mathcal{R} M'$ *or (iii)* $M' \longrightarrow M''$, $P \longrightarrow P_1$ *and* $P_1 \mathcal{R} M''$.

Weak barbed bisimilarity ${}_t{\approx_{c\pi}}$ *is the largest weak barbed bisimulation.*

The main peculiarities of the definition above are in condition 3, which captures the need of redoing some reductions that are unduly rolled-back in TransCCS, and in case (iii) of condition 5, which forces atomic abort.

Theorem 3. *For each TransCCS process* P, $P \, {}_t{\approx_{c\pi}} \, \nu k.\, k : (\!|P|\!)_t$.

Proof. The proof has to take into account the fact that different croll-π configurations may correspond to the same TransCCS process. In particular, a TransCCS transaction $[\![P \triangleright_k Q]\!]$ is matched in different ways if Q is the original compensation or if part of it is the result of an application of rule (R-Emb).

Thus, in the proof, we give a syntactic characterization of the set of croll-π configurations $(\!|P|\!)^p$ matching a TransCCS process P. Then we show that $\nu k.\, k : (\!|P|\!)_t \in (\!|P|\!)^p$, and that there is a match between reductions of P and the weak reductions of each configuration in $(\!|P|\!)^p$. The proof, in the two directions, is by induction on the rule applied to derive a single step. □

7 Related Work and Conclusion

We have presented a concurrent process calculus with explicit rollback and minimal facilities for alternatives built on a reversible substrate analogous to a Lévy labeling [4] for concurrent computations. We have shown by way of examples how to build more complex alternative idioms and how to use rollback and alternatives in conjunction to encode transactional constructs. In particular, we have developed an analysis of communicating transactions proposed in TransCCS [13]. We also developed a proof-of-concept interpreter of our language and used it to give a concurrent solution of the Eight Queens problem.

Undo or rollback capabilities in sequential languages have a long history (see [19] for an early survey). In a concurrent setting, interest has developed more recently. Works such as [9] introduce logging and process group primitives as a basis for defining fault-tolerant abstractions, including transactions. Ziarek et al. [26] introduce a checkpoint abstraction for concurrent ML programs. Field et al. [15] extend the actor model with checkpointing constructs. Most of the approaches relying instead on a fully reversible concurrent language have already been discussed in the introduction. Here we just recall that models of reversible computation have also been studied in the context of computational biology, e.g., [8]. Also, the effect of reversibility on Hennessy-Milner logic has been studied

in [23]. Several recent works have proposed a formal analysis of transactions, including [13] studied in this paper, as well as several other works such as [21, 5, 7] (see [1] for numerous references to the line of work concentrating on software transactional memories). Note that although reversible calculi can be used to implement transactions, they offer more flexibility. For instance, transactional events [14] only allow an all-or-nothing execution of transactions. Moreover, no visible side-effect is allowed during the transaction, as there is no way to specify how to compensate the side-effects of a failed transaction. A reversible calculus with alternatives allows the encoding of such compensations.

With the exception of the seminal work by Danos and Krivine [12] on RCCS, we are not aware of other work exploiting precise causal information as provided by our reversible machinery to analyze recovery-oriented constructs. Yet this precision seems important: as we have seen in Section 6, it allows us to weed out spurious undo of actions that appear in an approach that relies on a cruder transaction "embedding" mechanism. Although we have not developed a formal analysis yet, it seems this precision would be equally important, e.g., to avoid uncontrolled cascading rollbacks (domino effect) in [26] or to ensure that, in contrast to [15], rollback is always possible in failure-free computations. Although [9] introduces primitives able to track down causality information among groups of processes, called conclaves, it does not provide automatic support for undoing the effects of aborted conclaves, while our calculus directly provides a primitive to undo all the effects of a communication.

While encouraging, our results in Section 6 are only preliminary. Our concurrent rollback and minimal facilities for alternatives provide a good basis for understanding the "all-or-nothing" property of transactions. To this end it would be interesting to understand whether we are able to support both strong and weak atomicity of [21]. How to support isolation properties found, e.g., in software transactional memory models, in a way that combines well with these facilities remains to be seen. Further, we would like to study the exact relationships that exist between these facilities and the different notions of compensation that have appeared in formal models of computation for service-oriented computing, such as [5, 7]. It is also interesting to compare with zero-safe Petri nets [6], since tokens in zero places dynamically define transaction scopes as done by communications in croll-π.

From a practical point of view, we want both to refine the interpreter, and to test it against a wider range of more complex case studies. Concerning the interpreter, a main point is to allow for garbage collection of memories which cannot be restored any more, so to improve space efficiency.

References

[1] M. Abadi and T. Harris. Perspectives on transactional memory. In *CONCUR'09*, volume 5710 of *LNCS*. Springer, 2009.

[2] G. Bacci, V. Danos, and O. Kammar. On the statistical thermodynamics of reversible communicating processes. In *CALCO 2011*, volume 6859 of *LNCS*, 2011.

[3] W. W. Rouse Ball. *Mathematical Recreations and Essays (12th ed.)*. Macmillan, New York, 1947.
[4] G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *J. ACM*, 26(1), 1979.
[5] R. Bruni, H. C. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. In *POPL'05*. ACM, 2005.
[6] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2), 2000.
[7] M. J. Butler, C. A. R. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *25 Years CSP*, number 3525 in LNCS. Springer, 2004.
[8] L. Cardelli and C. Laneve. Reversible structures. In *CMSB 2011*. ACM, 2011.
[9] T. Chothia and D. Duggan. Abstractions for fault-tolerant global computing. *Theor. Comput. Sci.*, 322(3), 2004.
[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comp. Sci.*, 285(2), 2002.
[11] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR'04*, volume 3170 of *LNCS*. Springer, 2004.
[12] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.
[13] E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
[14] K. Donnelly and M. Fluet. Transactional events. *Journal of Functional Programming*, 18(5–6), 2008.
[15] J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL'05*. ACM, 2005.
[16] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Concurrent flexible reversibility (TR). `http://www.cs.unibo.it/~lanese/publications/fulltext/TR-crollpi.pdf.gz`, 2012.
[17] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR 2011*, volume 6901 of *LNCS*. Springer, 2011.
[18] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR 2010*, volume 6269 of *LNCS*. Springer, 2010.
[19] G. B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1), 1986.
[20] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In *FMOODS/FORTE 2012*, volume 7273 of *LNCS*, 2012.
[21] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL'08*. ACM, 2008.
[22] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2), 2007.
[23] I. Phillips and I. Ulidowski. A logic with reverse modalities for history-preserving bisimulations. In *EXPRESS 2011*, volume 64 of *EPTCS*, 2011.
[24] I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *RC 2012*, volume 7581 of *LNCS*, 2012.
[25] D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
[26] L. Ziarek and S. Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2), 2010.

Duality and i/o-types in the π-calculus

Daniel Hirschkoff[1], Jean-Marie Madiot[1], and Davide Sangiorgi[2]

[1] ENS Lyon, Université de Lyon, CNRS, INRIA, France,
[2] INRIA/Università di Bologna, Italy

Abstract. We study duality between input and output in the π-calculus. In dualisable versions of π, including πI and fusions, duality breaks with the addition of ordinary input/output types. We introduce $\overline{\pi}$, intuitively the minimal symmetrical conservative extension of π with input/output types. We prove some duality properties for $\overline{\pi}$ and we study embeddings between $\overline{\pi}$ and π in both directions. As an example of application of the dualities, we exploit the dualities of $\overline{\pi}$ and its theory to relate two encodings of call-by-name λ-calculus, by Milner and by van Bakel and Vigliotti, syntactically quite different from each other.

1 Introduction

It is common in mathematics to look for dualities; dualities may reveal underlying structure and lead to simpler theories. In turn, dualities can be used to relate different mathematical entities. In this work, our goal is to study dualities in the typed π-calculus, and to exploit them to understand the possible relationships between encodings of functions as π-calculus processes.

Reasoning about processes usually involves proving behavioural equivalences. In the case of the π-calculus, there is a well-established theory of equivalences and proof techniques. In some cases, it is necessary to work in a *typed* setting. Types allow one to express constraints about the observations available to the context when comparing two processes. One of the simplest and widely used such discipline is given by input/output-types [SW01] — i/o-types in the sequel.

In the π-calculus (simply called π below), the natural form of duality comes from the symmetry between input and output. There are several variants of π where processes can be 'symmetrised' by replacing inputs with outputs and vice versa. The π-calculus with internal mobility, πI [San96], is a subcalculus of π where only bound outputs are allowed (a bound output, that we shall note $\overline{a}(x).P$, is the emission of a private name x on some channel a). In πI, duality can be expressed at an operational level, by exchanging (bound) inputs and bound outputs: the dual of $a(x).\overline{x}(y).0$ is $\overline{a}(x).x(y).0$.

Other well-known variants of π with dualities are the calculi in the fusion family [PV98,Fu97,GW00]. In fusions, a construct for *free input* acts as the dual of the free output construct of π, and the calculus has only one binder, restriction. Interaction on a given channel has the effect of *fusing* (that is, identifying) names.

The discipline of simple types can be adapted both to πI and to fusions, while preserving dualities. The situation is less clear for i/o-types, which can

be very useful to establish equivalences between processes. Let us give some intuitions about why it is so. In i/o-types, types are assigned to channels and express *capabilities*: a name of type oT can be used only to emit values of type T, and similarly for the input capability (iT). This is expressed by the following typing rules for i/o-types in π:

$$\frac{\Gamma \vdash a : iT \qquad \Gamma, x : T \vdash P}{\Gamma \vdash a(x).P} \qquad\qquad \frac{\Gamma \vdash a : oT \qquad \Gamma \vdash b : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}b.P}$$

The rule for input can be read as follows: process $a(x).P$ is well-typed provided (i) the typing environment, Γ, ensures that the input capability on a can be derived, and (ii) the continuation of the input can be typed in an environment where x is used according to T. The typing rule for output checks that (i) the output capability on a is derivable, (ii) the emitted value, b, has the right type, and (iii) the continuation P can be typed. As an example, $a : i(iT) \vdash a(x).\overline{x}t.0$ cannot be derived, because only the input capability is received on a, which prevents $\overline{x}t.0$ from being typable.

I/o-types come with a notion of subtyping, that makes it possible to relate type $\sharp T$ (which stands for both input and output capabilities) with input and output capabilities (in particular, we have $\sharp T \leq iT$ and $\sharp T \leq oT$). We stress an asymmetry between the constraints attached to the transmitted name in the two rules above. Indeed, while in a reception we somehow enforce a "contract" on the usage of the received name, in the rule for output this is not the case: we can use subtyping in order to derive type, say, iU for b when typechecking the output, while b's type can be $\sharp U$ when typechecking the continuation P.

The starting point of this work is the conflict between the asymmetry inherent to i/o-types and the symmetries we want to obtain via duality. For example i/o-types can be adapted to πI, but duality cannot be applied to the resulting typings. In fusion calculi, the conflict with the asymmetry of i/o-types is even more dramatic. Indeed, subtyping in i/o-types is closely related to substitution, since replacing a name with another makes sense only if the latter has a more general type. Fusions are intuitively substitutions operating in both directions, which leaves no room for subtyping. In work in preparation [HMS12], we investigate this relationship between subtyping and substitution, and compare several variants of existing calculi, including the one presented in this paper.

In this paper, in order to work in a setting that provides a form of duality and where i/o-types can be used, we introduce a calculus named $\overline{\pi}$ (Section 2). $\overline{\pi}$ is an extension of π with constructs for free input and bound output (note that bound output is not seen as a derived construct in $\overline{\pi}$). In $\overline{\pi}$, we rely on substitutions as the main mechanism at work along interactions. To achieve this, we forbid interactions involving a free input and a free output: the type system rules out processes that use both kinds of prefixes on the same channel.

Calculus $\overline{\pi}$ contains π, and any π process that can be typed using i/o-types can be typed in exactly the same way in $\overline{\pi}$. Moreover $\overline{\pi}$ contains a 'dualised' version of π: one can choose to use some channels in free input and bound output.

For such channels, the typing rules intuitively enforce a 'contract' on the usage of the transmitted name *on the side of the emitter* (dually to the typing rules presented above). We show how $\overline{\pi}$ can be related to π, by translating $\overline{\pi}$ into a variant of the π-calculus with i/o-types in a fully abstract way. This result shows that π and $\overline{\pi}$ are rather close in terms of expressiveness.

We also define a notion of typed barbed congruence in $\overline{\pi}$, which allows us to validate at a behavioural level the properties we have mentioned above: two processes are equivalent if and only if their duals are. To our knowledge, no existing calculus with i/o-types enjoys this form of duality for behaviours.

As an application of $\overline{\pi}$, its dualities, and its behavioural theory, we use $\overline{\pi}$ to relate two encodings of call-by-name λ-calculus. The first one is the ordinary encoding by Milner [Mil92], the second one is by van Bakel and Vigliotti [vBV09]. The two encodings are syntactically quite different. Milner's is *input-based*, in that an abstraction interacts with its environment via an input. In contrast, van Bakel and Vigliotti's is *output-based*. Moreover, only the latter makes use of *link processes*, that is, forwarders that under certain conditions act as substitutions.

Van Bakel and Vigliotti actually encode *strong* call-by-name — reductions may also take place inside a λ-abstraction. We therefore compare van Bakel and Vigliotti's encoding with the strong variant of Milner's encoding, obtained by replacing an input with a delayed input, following [Mer00] (in a delayed input $a(x){:}P$, the continuation P may perform transitions not involving the binder x even when the head input at a has not been consumed).

We exploit $\overline{\pi}$ (in fact the extension of $\overline{\pi}$ with delayed input) to prove that the two encodings are the dual of one another. This is achieved by first embedding the π-terms of the λ-encodings into $\overline{\pi}$, and then applying behavioural laws of $\overline{\pi}$. The correctness of these transformations is justified using i/o-types (essentially to express the conditions under which a link can be erased in favour of a substitution). Some of the transformations needed for the λ-encodings, however, are proved in this paper only for barbed bisimilarity; see the concluding section for a discussion.

Paper outline. Section 2 introduces $\overline{\pi}$, and presents its main properties. To analyse dualities in encodings of λ into π, in Section 3, we extend $\overline{\pi}$, notably with delayed prefixes. As the addition of these constructs is standard, they are omitted from the original syntax so to simplify the presentation. Section 4 gives concluding remarks.

2 $\overline{\pi}$, a symmetric π-calculus

In this section, we present $\overline{\pi}$, a π-calculus with i/o-types that enjoys duality properties. We define the syntax and operational semantics for $\overline{\pi}$ processes in Section 2.1, introduce types and barbed congruence in Section 2.2, establish duality in Section 2.3, and present results relating π and $\overline{\pi}$ in Section 2.4.

2.1 Syntax and Operational Semantics

We consider an infinite set of names, ranged over using $a, b, \ldots, x, y, \ldots$. The syntax of $\overline{\pi}$ is as follows:

$$P ::= 0 \mid P|P \mid !P \mid \alpha.P \mid (\nu a)P \qquad \alpha ::= \rho b \mid \rho(x) \qquad \rho ::= a \mid \overline{a}$$

$\overline{\pi}$ differs from the usual π-calculus by the presence of the free input ab and bound output $\overline{a}(x)$ prefixes. Note that in $\overline{\pi}$, the latter is *not* a notation for $(\nu x)\overline{a}x.P$, but a primitive construct. These prefixes are the symmetric counterpart of $\overline{a}b$ and $a(x)$ respectively. Given a process P, $\mathrm{fn}(P)$ stands for the set of free names of P — restriction, bound input and bound output are binding constructs. Given ρ of the form a or $\overline{a}$, $\mathrm{n}(\rho)$ is defined by $\mathrm{n}(\overline{a}) = \mathrm{n}(a) = a$.

Structural congruence is standard, and defined as in π (in particular, there are no axioms involving prefixes). The reduction laws allow communication involving two prefixes *only if at least one of them is bound*:

$$\begin{array}{rr}
\overline{a}b.P \mid a(x).Q \to P \mid Q[b/x] & P \to Q \text{ if } P \equiv \to \equiv Q \\
ab.P \mid \overline{a}(x).Q \to P \mid Q[b/x] & (\nu a)P \to (\nu a)Q \text{ if } P \to Q \\
\overline{a}(x).P \mid a(x).Q \to (\nu x)(P \mid Q) & P \mid R \to Q \mid R \text{ if } P \to Q
\end{array}$$

Note that $\overline{a}b \mid ac$ is a process of $\overline{\pi}$ that has no reduction; this process is ruled out by the type system presented below.

2.2 Types and Behavioural Equivalence

Types are a refinement of standard i/o-types: in addition to capabilities (ranged over using c), we annotate types with *sorts* (s), that specify whether a name can be used in free input (sort $\mathbf{e}$) or in free output ($\mathbf{r}$) — note that a name cannot be used to build both kinds of free prefixes.

$$T ::= c^s T \mid \mathbf{1} \qquad c ::= i \mid o \mid \sharp \qquad s ::= \mathbf{e} \mid \mathbf{r}$$

If name a has type $c^{\mathbf{r}}T$, we shall refer to a as an $\mathbf{r}$-*name*, and similarly for $\mathbf{e}$.

The subtyping relation is the smallest reflexive and transitive relation $\leq$ satisfying the rules of Figure 1. As in the π-calculus $i^{\mathbf{r}}$ is covariant and $o^{\mathbf{r}}$ is contravariant. Dually, $i^{\mathbf{e}}$ is contravariant and $o^{\mathbf{e}}$ is covariant. Note that sorts ($\mathbf{e}$, $\mathbf{r}$) are not affected by subtyping.

The type system is defined as a refinement of input/output types, and is given by the rules of Figure 2. There is a dedicated typing rule for every kind of prefix (free, ρb, or bound, $\rho(x)$), according to the sort of the involved name. We write $\Gamma(a)$ for the type associated to a in Γ. $T^{\leftrightarrow}$ stands for T where we switch the top-level capability: $(c^s T)^{\leftrightarrow} = \overline{c}^s T$ where $\overline{o} = i, \overline{i} = o, \overline{\sharp} = \sharp$. The typing rules for $\mathbf{r}$-names impose a constraint on the $\mathbf{r}$eceiving side: all inputs on an $\mathbf{r}$-channel should be bound. Note that $\overline{a}(x).P$ and $(\boldsymbol{\nu} x)\overline{a}x.P$ are *not* equivalent from the point of view of typing: typing a bound output on an $\mathbf{r}$-channel (a) imposes that the transmitted name (x) is used according to the "dual constraint" w.r.t.

$$\frac{}{\sharp^s T \leq i^s T} \qquad \frac{}{\sharp^s T \leq o^s T}$$

$$\frac{T_1 \leq T_2}{i^{\mathbf{r}} T_1 \leq i^{\mathbf{r}} T_2} \qquad \frac{T_1 \leq T_2}{o^{\mathbf{r}} T_2 \leq o^{\mathbf{r}} T_1} \qquad \frac{T_1 \leq T_2}{i^{\mathbf{e}} T_2 \leq i^{\mathbf{e}} T_1} \qquad \frac{T_1 \leq T_2}{o^{\mathbf{e}} T_1 \leq o^{\mathbf{e}} T_2}$$

Fig. 1. Subtyping

$$\frac{\Gamma \vdash a : i^{\mathbf{r}} T \qquad \Gamma, x : T \vdash P}{\Gamma \vdash a(x).P} \qquad \frac{\Gamma \vdash a : i^{\mathbf{e}} T \qquad \Gamma, x : T^{\leftrightarrow} \vdash P}{\Gamma \vdash a(x).P}$$

$$\frac{\Gamma \vdash a : o^{\mathbf{e}} T \qquad \Gamma, x : T \vdash P}{\Gamma \vdash \overline{a}(x).P} \qquad \frac{\Gamma \vdash a : o^{\mathbf{r}} T \qquad \Gamma, x : T^{\leftrightarrow} \vdash P}{\Gamma \vdash \overline{a}(x).P}$$

$$\frac{\Gamma \vdash a : i^{\mathbf{e}} T \qquad \Gamma \vdash b : T \qquad \Gamma \vdash P}{\Gamma \vdash ab.P} \qquad \frac{\Gamma \vdash a : o^{\mathbf{r}} T \qquad \Gamma \vdash b : T \qquad \Gamma \vdash P}{\Gamma \vdash \overline{a}b.P}$$

$$\frac{\Gamma, a : T \vdash P}{\Gamma \vdash (\nu a)P} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \qquad \frac{\Gamma \vdash P}{\Gamma \vdash !P} \qquad \frac{}{\Gamma \vdash 0} \qquad \frac{\Gamma(a) \leq T}{\Gamma \vdash a : T}$$

Fig. 2. $\overline{\pi}$: Typing rules

what a's type specifies: this is enforced using $T^{\leftrightarrow}$ (while names received on a are used according to T). Symmetrical considerations can be made for **e**-names, that impose constraints on the **e**mitting side.

We write $\Gamma \vdash P, Q$ when both $\Gamma \vdash P$ and $\Gamma \vdash Q$ can be derived.

Remark 1 ("Double contract"). We could adopt a more liberal typing for bound outputs on **r** names, and use the rule

$$\frac{\Gamma \vdash a : o^{\mathbf{r}} T \qquad \Gamma, x : T' \vdash P \qquad T' \leq T}{\Gamma \vdash \overline{a}(x).P}$$

(and its counterpart for inputs on **e**-names). This would have the effect of typing $\overline{a}(x).P$ like $(\boldsymbol{\nu} x)\overline{a}x.P$. We instead chose to enforce what we call a *"double contract"*: the same way a receiving process uses the bound name according to the type specified in the channel that is used for reception, the continuation of a bound output uses the emitted name according to $T^{\leftrightarrow}$, the *symmetrised version* of T. This corresponds to a useful programming idiom in π, where it is common to create a name, transmit one capability on this name and use locally the other, dual capability. This idiom is used e.g. in [Vas09] and in [SW01, Sect. 5.7.3]. This choice moreover makes the proofs in Section 3.2 easier.

Observe that when a typable process reduces according to

$$\overline{a}(x).P \mid a(x).Q \rightarrow (\nu x)(P \mid Q) \ ,$$

if a has type, say, $\sharp^{\mathbf{r}}(o^s T)$, then in the right hand side process, name x is given type $\sharp^s T$, and the $\sharp$ capability is "split" into $i^s T$ (used by P) and $o^s T$ (used by Q) — it would be the other way around if a's sort were $\mathbf{e}$.

Lemma 1 (Properties of typing).

1. *(Weakening) If* $\Gamma \vdash P$ *then* $\Gamma, a : T \vdash P$.
2. *(Strengthening) If* $\Gamma, a : T \vdash P$ *and* $a \notin fn(P)$ *then* $\Gamma \vdash P$.
3. *(Narrowing) If* $\Delta \leq \Gamma$ *and* $\Gamma \vdash P$ *then* $\Delta \vdash P$.
4. *(Substitution) If* $\Gamma, x : T \vdash P$ *and* $\Gamma \vdash b : T$ *then* $\Gamma \vdash P[b/x]$.

Proposition 1 (Subject reduction). *If* $\Gamma \vdash P$ *and* $P \to Q$ *then* $\Gamma \vdash Q$.

Proof. By transition induction. Lemma 1 (4) is used when a bound prefix communicates with a free prefix; Lemma 1 (3) is used for the interaction between two bound prefixes, since T and $T^{\leftrightarrow}$ have a common subtype. □

Definition 1 (Contexts). *Contexts are processes with one occurrence of the hole, written* $[-]$. *They are defined by the following grammar:*

$$C ::= [-] \mid C|P \mid P|C \mid !C \mid \alpha.C \mid (\nu a)C \ .$$

Definition 2. *Let* Γ, Δ *be typing environments. We say that* Γ extends Δ *if the support of* Δ *is included in the support of* Γ, *and if* $\Delta \vdash x : T$ *entails* $\Gamma \vdash x : T$ *for all* x. *A context* C *is a* (Γ/Δ)*-context, written* $\Gamma/\Delta \vdash C$, *if* C *can be typed in the environment* Γ, *the hole being well-typed in any context that extends* Δ.

As a consequence of the previous definition and of Lemma 1, it is easy to show that if $\Delta \vdash P$ and $\Gamma/\Delta \vdash C$, then $\Gamma \vdash C[P]$.

We now move to the definition of behavioural equivalence.

Definition 3 (Barbs). *Given* $\rho \in \{a, \overline{a}\}$, *where* a *is a name, we say that* P *exhibits barb* ρ, *written* $P{\downarrow_\rho}$, *if* $P \equiv (\nu c_1 \dots c_n)(\alpha.Q \mid R)$ *where* $\alpha \in \{\rho(x), \rho b\}$ *with* $a \notin \{c_1, \dots, c_n\}$. *We extend the definition to* weak barbs*:* $P \Downarrow_\rho$ *stands for* $P \Rightarrow{\downarrow_\rho}$ *where* $\Rightarrow$ *is the reflexive transitive closure of* $\to$.

Definition 4 (Typed barbed congruence). Barbed bisimilarity *is the largest symmetric relation* $\dot{\approx}$ *such that whenever* $P \dot{\approx} Q$, $P \downarrow_\rho$ *implies* $Q \Downarrow_\rho$ *and* $P \to P'$ *implies* $Q \Rightarrow \dot{\approx} P'$. *When* $\Delta \vdash P, Q$, *we say that* P *and* Q *are* barbed congruent at Δ, *written* $\Delta \rhd P \cong^c Q$, *if for all* (Γ/Δ)*-context* C, $C[P] \dot{\approx} C[Q]$.

2.3 Duality

Definition 5 (Dual of a process). *The dual of a process* P, *written* $\overline{P}$, *is the process obtained by transforming prefixes as follows:* $\overline{\overline{a}b} = ab$, $\overline{ab} = \overline{a}b$, $\overline{\overline{a}(x)} = a(x)$, $\overline{a(x)} = \overline{a}(x)$, *and applying dualisation homeomorphically to the other constructs.*

Lemma 2 (Duality for reduction). *If* $P \to Q$ *then* $\overline{P} \to \overline{Q}$.

Dualising a type means swapping i/o capabilities and $\mathbf{e}/\mathbf{r}$ sorts.

Definition 6 (Dual of a type). *The dual of T, written $\overline{T}$, is defined by setting $\overline{c^s T} = \overline{c}^{\overline{s}}\,\overline{T}$, with $\overline{\mathbf{r}} = \mathbf{e}, \overline{\mathbf{e}} = \mathbf{r}, \overline{i} = o$, and $\overline{o} = i$. We extend the definition to typing environments, and write $\overline{\Gamma}$ for the dual of Γ.*

Lemma 3 (Duality for typing).

1. *If $T_1 \leq T_2$ then $\overline{T_1} \leq \overline{T_2}$.*
2. *If $\Gamma \vdash P$ then $\overline{\Gamma} \vdash \overline{P}$.*
3. *If $\Gamma/\Delta \vdash C$ then $\overline{\Gamma}/\overline{\Delta} \vdash \overline{C}$.*

Proof. (1): the covariant type operators ($i^{\mathbf{r}}$ and $o^{\mathbf{e}}$) are dual of each other, and so are the contravariant operators ($o^{\mathbf{r}}$ and $i^{\mathbf{e}}$). (2) follows from the shape of the typing rules, e.g., the dual of the rule for $i^{\mathbf{r}}$ is an instance of the rule for $\overline{i^{\mathbf{r}}} = o^{\mathbf{e}}$. (3) holds because if Φ extends Δ then $\overline{\Phi}$ extends $\overline{\Delta}$ (item (1)). □

Most importantly, duality holds for typed barbed congruence. The result is easy in the untyped case, since duality preserves reduction and dualises barbs. On the other hand, we are not aware of the existence of another system having this property in presence of i/o-types.

Theorem 1 (Duality for $\cong^c$). *If $\Delta \rhd P \cong^c Q$ then $\overline{\Delta} \rhd \overline{P} \cong^c \overline{Q}$.*

Proof. By Lemma 3, we only have to prove that if $P \mathrel{\dot{\approx}} Q$ then $\overline{P} \mathrel{\dot{\approx}} \overline{Q}$, i.e., duality preserves reduction and swaps barbs. □

2.4 Embeddings between π and $\overline{\pi}$

From $\overline{\pi}$ to π^{io}. As explained in Section 1, the π-calculus with i/o-types (that we note π^{io}) is an asymmetric calculus. In some sense, $\overline{\pi}$ can be seen as a 'dualisation' of π^{io}. This can be formulated rigorously by projecting $\overline{\pi}$ into π^{io}. To define this projection, which we call a *partial dualisation*, we work in an extended version of π^{io}, where capabilities are duplicated: in addition to the $i, o, \sharp$ capabilities, we also have capabilities $\underline{i}, \underline{o}$ and $\underline{\sharp}$, that intuitively correspond to the image of the "$\mathbf{e}$-part" of $\overline{\pi}$ through the encoding. The additional capabilities act exactly like the corresponding usual capabilities, in particular w.r.t. subtyping and duality. We write π_2^{io} for the resulting calculus. We discuss below (Remark 3) to what extent the addition of these capabilities is necessary. We also rely on π_2^{io} to prove that $\overline{\pi}$ is a conservative extension of the π-calculus in Theorem 2 — π_2^{io} is actually close, operationally, to both calculi.

Definition 7 (Partial dualisation). *We define a translation from typed processes in $\overline{\pi}$ to π_2^{io}. The translation acts on typing derivations: given a derivation δ of $\Gamma \vdash P$ (written $\delta :: \Gamma_\delta \vdash P$), we define a π_2^{io} process noted $[P]^\delta$ as follows:*

$$
\begin{array}{rl}
[\rho b.P]^\delta = \overline{\rho} b.[P]^{\delta'} & \text{if } \Gamma_\delta(\mathrm{n}(\rho)) = c^{\mathbf{e}}T \\
[\rho b.P]^\delta = \rho b.[P]^{\delta'} & \text{if } \Gamma_\delta(\mathrm{n}(\rho)) = c^{\mathbf{r}}T \\
[\rho(x).P]^\delta = \overline{\rho}(x).[P]^{\delta'} & \text{if } \Gamma_\delta(\mathrm{n}(\rho)) = c^{\mathbf{e}}T \\
[\rho(x).P]^\delta = \rho(x).[P]^{\delta'} & \text{if } \Gamma_\delta(\mathrm{n}(\rho)) = c^{\mathbf{r}}T
\end{array}
$$

$$
[(\nu a)P]^\delta = [P]^{\delta'} \qquad [0]^\delta = 0 \qquad [!P]^\delta = ![P]^{\delta'} \qquad [P \mid Q]^\delta = [P]^{\delta'_1} \mid [Q]^{\delta'_2}
$$

In the above definition, δ' is the subderivation of δ, in case there is only one, and δ'_1 and δ'_2 are the obvious subderivations in the case of parallel composition. We extend the definition to types: T^* stands for T where all occurrences of $c^{\mathbf{r}}$ (resp. $c^{\mathbf{e}}$) are replaced with c (resp. $\overline{c}$, the dual of $\underline{c}$). We define accordingly Γ^*.

Remark 2. The same translation could be defined for a simply typed version of $\overline{\pi}$. Indeed, $[\cdot]^{\text{-}}$ does not depend on capabilities ($i/o/\sharp$), but only on sorts ($\mathbf{r}/\mathbf{e}$).

Lemma 4. *If $\delta :: \Gamma \vdash P$ (in $\overline{\pi}$), then $\Gamma^* \vdash [P]^\delta$ (in π_2^{io}).*

Proof. In moving from Γ to Γ^*, we replace $i^{\mathbf{e}}$ (resp. $o^{\mathbf{e}}$, $i^{\mathbf{r}}$, $o^{\mathbf{r}}$) with $\underline{o}$ (resp. $\underline{i}$, i, o). This transformation preserves the subtyping relation. Moreover, the rules to type prefixes $i^{\mathbf{r}}, o^{\mathbf{r}}, i^{\mathbf{e}}, o^{\mathbf{e}}$ in $\overline{\pi}$ correspond to the rules for $i, o, \underline{o}, \underline{i}$ in π_2^{io}. □

Lemma 5. *Whenever $\delta_1 :: \Gamma \vdash P$ and $\delta_2 :: \Gamma \vdash P$, we have $\Gamma^* \rhd [P]^{\delta_1} \simeq^c [P]^{\delta_2}$.*

Proof. The relation $\mathcal{R} \triangleq \{([P]^{\delta_1}, [P]^{\delta_2}) \mid \delta_1, \delta_2 :: \Gamma \vdash P\}$ is a strong bisimulation in π and is substitution-closed; hence $\mathcal{R}$ is included in $\simeq^c$, since $[P]^{\delta_i}$ is typable in Γ^* (by Lemma 4). □

Lemma 6. *If $\delta_P :: \Gamma \vdash P$ and $\delta_Q :: \Gamma \vdash Q$ then we have the following:*

1. *(P and Q have the same barbs) iff ($[P]^{\delta_P}$ and $[Q]^{\delta_Q}$ have the same barbs)*
2. *if $P \to P'$ then $[P]^{\delta_P} \to [P']^{\delta}$ for some $\delta :: \Gamma \vdash P'$.*
3. *if $[P]^{\delta_P} \to P_1$ then $P_1 = [P']^{\delta}$ with $P \to P'$ for some $\delta :: \Gamma \vdash P'$.*
4. *$P \dot{\approx} Q$ iff $[P]^{\delta_P} \dot{\approx} [Q]^{\delta_Q}$.*

Proof. (4) is a consequence of (1), (2), (3). For (1) remark that if $\Gamma(a) = c^{\mathbf{r}}T$ then P and $[P]^{\delta_P}$ have the same barbs on a; if $\Gamma(a) = c^{\mathbf{e}}T$, they have dual barbs on a, but in this case so do Q and $[Q]^{\delta_Q}$. For (2) and (3), we remark that $[\cdot]^\delta$ is compositional and preserves the fact that two prefixes can interact — even when moving to a different δ. □

Proposition 2 (Full abstraction). *If $\delta_P :: \Gamma \vdash P$ and $\delta_Q :: \Gamma \vdash Q$ then*

$$\Gamma \rhd P \cong^c Q \text{ (in } \overline{\pi}) \quad \text{iff} \quad \Gamma^* \rhd [P]^{\delta_P} \cong^c [Q]^{\delta_Q} \text{ (in } \pi_2^{\text{io}}) \ .$$

Proof. Soundness: given a derivation $\gamma :: \Delta/\Gamma \vdash C$, we build $[C]^\gamma$ which is a (Δ^*/Γ^*)-context. Then $[C]^\gamma[[P]^{\delta_P}] = [C[P]]^{\beta_P}$ for some β_P and we can rely on barbed congruence in π_2^{io} to establish $[C[P]]^{\beta_P} \dot{\approx} [C[Q]]^{\beta_Q}$. By Lemma 6, we deduce $C[P] \dot{\approx} C[Q]$.
Completeness: we define the reverse translation $\{\cdot\}^{\text{-}}$ of $[\cdot]^{\text{-}}$ and reason as above to prove its soundness. Thanks to the fact that $\delta_P :: \Gamma \vdash P$ implies $\{[P]^{\delta_P}\}^{\delta_P^*} = P$ where $\delta_P^* :: \Gamma^* \vdash [P]^\delta$ is the derivation obtained by Lemma 4, the soundness of $\{\cdot\}^{\text{-}}$ implies the completeness of $[\cdot]^{\text{-}}$, and vice versa. □

Remark 3 (π_2^{io} vs π^{io}). We can make two remarks about the above result.

First, it would seem natural to project directly onto π^{io}, by mapping capabilities $i^{\mathbf{r}}$ and $o^{\mathbf{e}}$ into i, and $o^{\mathbf{r}}$ and $i^{\mathbf{e}}$ into o. However, the result of Proposition 2

would not hold in this case. The intuitive reason is that in doing so, we would allow two names having different sorts in $\overline{\pi}$ to be equated in the image of the encoding, thus giving rise to additional observations (since we cannot equate names having different sorts in $\overline{\pi}$). Technically, this question is reminiscent of the problem of closure of bisimilarity under substitutions in the π-calculus.

Second, the key ingredient in the definition of partial dualisation is to preserve the distinction between names having originally different sorts in the $\overline{\pi}$ process. It is possible to define an encoding of π_2^{io} into *a dyadic version* of π^{io} (without the extra capabilities), in order to do so.

Lemma 7. *Suppose $\Delta \vdash P, Q$ holds in π^{io}.*
Then $\Delta \rhd P \cong^c Q$ (in π^{io}) iff $\Delta \rhd P \cong^c Q$ (in π_2^{io}).

Proof. The right-to-left implication is immediate because any π^{io}-context is a π_2^{io}-context. To show the converse, we observe that a (Γ/Δ)-context in π_2^{io} is a (Γ'/Δ)-context in π^{io}, where Γ' is Γ where every $\underline{c}$ capability is replaced with c.

From π^{io} to $\overline{\pi}$. $\overline{\pi}$ contains π^{io}, the π-calculus with i/o-types: the rules for $\mathbf{r}$-channels are exactly those of π^{io}, and typability of $\mathbf{e}$-free processes coincides with typability in π^{io}. More precisely we can say that $\overline{\pi}$ is a conservative extension of π^{io}. In π^{io} we rely on typed barbed congruence as defined in [SW01], which is essentially the same as $\cong^c$ in $\overline{\pi}$. Before presenting the result, the following remark introduces some notation.

Remark 4. Suppose $\delta :: \Gamma \vdash P$, in π^{io}. Then $\delta^{\mathbf{r}} :: \Gamma^{\mathbf{r}} \vdash P$ in $\overline{\pi}$, where $\Gamma^{\mathbf{r}}$ stands for Γ in which all types are decorated with $\mathbf{r}$ and $\delta^{\mathbf{r}}$ stands for δ where all usages of the typing rule for restriction introduce an $\mathbf{r}$-type. Moreover $[P]^{\delta^{\mathbf{r}}} = P$.

Theorem 2 (Conservative extension). *Suppose $\Gamma \vdash P, Q$ holds in π^{io}.*
Then $\Gamma \rhd P \cong^c Q$ (in π^{io}) iff $\Gamma^{\mathbf{r}} \rhd P \cong^c Q$ (in $\overline{\pi}$).

Proof. We use π_2^{io} as an intermediate calculus. By Remark 4, let δ_P, δ_Q be derivations of $\Gamma^{\mathbf{r}} \vdash P$ and $\Gamma^{\mathbf{r}} \vdash Q$ such that $P = [P]^{\delta_P}$ and $Q = [Q]^{\delta_Q}$. By Proposition 2, the right hand side is equivalent to $(\Gamma^{\mathbf{r}})^* \rhd [P]^{\delta_P} \cong^c [Q]^{\delta_Q}$ (in π_2^{io}). By hypothesis, and since $(\Gamma^{\mathbf{r}})^* = \Gamma$, the latter is equivalent to $\Gamma \rhd P \cong^c Q$ (in π_2^{io}). Lemma 7 allows us to finish the proof. □

The result above shows that π can be embedded rather naturally into $\overline{\pi}$. This is in contrast with fusion calculi, where the equivalence on π-calculus terms induced by the embedding into fusions does not coincide with a barbed congruence or equivalence in the π-calculus.

Remark 5 ($\overline{\pi}$ and existing symmetric calculi). $\overline{\pi}$ contains the π-calculus, and hence contains (the typed version of) πI, the π-calculus with internal mobility (see [SW01]). On the other hand, because free inputs and free outputs are not allowed to interact in $\overline{\pi}$, $\overline{\pi}$ fails to represent the fusion calculus. As mentioned above, we have not succeeded in defining a 'symmetrical version' of i/o-types that would be suitable for fusions.

3 Application: Relating Encodings of the λ-calculus

In this section, we use $\overline{\pi}$ to reason about encodings of the (call-by-name) λ-calculus into the π-calculus. To do so, we need to extend $\overline{\pi}$ (Section 3.1). We then justify the validity of a transformation that makes use of link processes in Section 3.2. Finally, we show how duality, together with the latter transformation, allows us to relate Milner's encoding with the one of van Bakel and Vigliotti.

3.1 Extending $\overline{\pi}$

Based on $\overline{\pi}$, we develop an extension, called $\overline{\pi}^{\mathbf{a}}$, with forms of asynchronous communication and polyadicity. The extension to polyadic communication is standard. Asynchronous communication is added via the inclusion of *delayed* prefixes: $a(x){:}P$ (resp. $\overline{a}(x){:}P$) stands for a *(bound) delayed input (resp. output)* prefix. The intuition behind delayed prefixes is that they allow the continuation of the prefix to interact, as long as the performed action is not causally dependent on the prefix itself — this is made more precise below. Intuitively asynchrony is useful when reasoning about encodings of the λ-calculus because in a β-reduction $(\lambda x.M)N \to M[N/x]$ the "output" part N has no continuation. It is also useful to have asynchrony in input because the considered λ-strategy allows reduction under a λ-abstraction. Moreover asynchrony allows us to derive some transformation laws involving link processes (Section 3.2). Note that synchronous prefixes are still necessary, to encode the argument of an application.

Delayed prefixes are typed like bound input and output prefixes in Section 2. Types are refined with two new sorts that enforce asynchrony: $\mathbf{d}$ to force inputs to be bound and delayed, $\mathbf{a}$ to force outputs to be bound and delayed — we call such outputs *asynchronous*. For instance, if we have $a : \sharp^{\mathbf{r}}_{\mathbf{d}} T$ for some T, then all inputs at a are bound and delayed. We also include recursive types.

$$T ::= c_t\langle {}^{s_1}T_1, \ldots, {}^{s_n}T_n\rangle \mid \mathbf{1} \mid \mu X.T \mid X \qquad s ::= \mathbf{e} \mid \mathbf{r} \qquad t ::= \mathbf{d} \mid \mathbf{a}$$

In the polyadic case, $\mathbf{e}/\mathbf{r}$ sorts are given to each element of the transmitted tuple. We present here only the typing rule for delayed input, in polyadic form, to illustrate how we extend the type system of Section 2.

$$\frac{\Gamma \vdash a : i_t\langle {}^{s_1}T_1, \ldots {}^{s_n}T_n\rangle \quad \Gamma, x_1 : T_1^{s_1}, \ldots, x_n : T_n^{s_n} \vdash P}{\Gamma \vdash a(x_1, \ldots, x_n){:}P}$$

(with $T^{\mathbf{r}} = T$ and $T^{\mathbf{e}} = T^{\leftrightarrow}$). The sort $\mathbf{d}$ (resp. $\mathbf{a}$) is forbidden in the rules to type non-delayed input (resp. output) prefixes.

The definition of operational semantics is extended as follows to handle delayed prefixes (below, $\overline{\rho}(y)P$ stands for either $\overline{\rho}(y).P$ or $\overline{\rho}(y){:}P$):

$$\begin{array}{rl} P \mid \rho(x){:}Q \equiv \rho(x){:}(P \mid Q) & \text{if } x \notin \mathrm{fn}(P) \\ \rho_1(y){:}\rho_2(x){:}P \equiv \rho_2(x){:}\rho_1(y){:}P & \text{if } \mathrm{n}(\rho_1) \neq x, x \neq y, y \neq \mathrm{n}(\rho_2) \\ (\nu y)\rho(x){:}P \equiv \rho(x){:}(\nu y)P & \text{if } x \neq y, y \neq \mathrm{n}(\rho) \end{array}$$

$$\rho(x){:}(\overline{\rho}(y)P \mid Q) \to (\nu y)(P \mid Q)[y/x] \qquad \rho(x){:}P \to \rho(x){:}Q \quad \text{if } P \to Q$$
$$\rho(x){:}(\overline{\rho}b.P \mid Q) \to (P \mid Q)[b/x]$$

Barbs are defined as in Section 2, with an additional clause saying that if ρ is a barb of P and $\mathrm{n}(\rho) \neq x$, then ρ is a barb of $\rho'(x){:}P$.

The results of Section 2 hold for this extended calculus, with similar proofs:

Proposition 3 (Duality, extended calculus).

1. *Duality of typing:* $\Gamma \vdash P \Rightarrow \overline{\Gamma} \vdash \overline{P}$.
2. *Duality of barbed congruence:* $\Gamma \rhd P \cong^c Q \Rightarrow \overline{\Gamma} \rhd \overline{P} \cong^c \overline{Q}$.

The counterpart of Theorem 2 also holds in $\overline{\pi}^{\mathbf{a}}$, which stands for the extended calculus of this section, where types also specify how names have to be used in delayed prefixes. It can be stated w.r.t. $\pi^{\mathrm{io},\mathbf{a}}$, which is defined as π^{io} with additional typing information to specify which names have to be used asynchronously.

Theorem 3 (Conservative extension, extended calculus). *Suppose we have* $\Gamma \vdash P, Q$ *in* $\pi^{\mathrm{io},\mathbf{a}}$. *Then* $\Gamma \rhd P \cong^c Q$ *(in* $\pi^{\mathrm{io},\mathbf{a}}$*) iff* $\Gamma^{\mathbf{r}} \rhd P \cong^c Q$ *(in* $\overline{\pi}^{\mathbf{a}}$*)* .

The extensions $\overline{\pi}^{\mathbf{a}}$ and $\pi^{\mathrm{io},\mathbf{a}}$ are asynchronous versions of $\overline{\pi}$ and π^{io} in the sense that interaction is no longer a synchronous handshaking between two processes: for at least one of the processes, the occurrence of the interaction is not observable because the consumed action is not blocking for a continuation.

3.2 Reasoning about Links, a transformation from $o^{\mathbf{e}}_{\mathbf{a}}$ to $i^{\mathbf{r}}_{\mathbf{a}}$

The main result of this section is a technical lemma about the validity of a transformation which is used for the analysis of λ-calculus encodings in Section 3.3. A reader not interested in this result may safely skip this section.

Differently from partial dualisation (Definition 7), the transformation, written $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$, modifies prefixes, beyond simple dualisation, by introducing link processes. It also acts on types, by mapping **e**-names onto **r**-names.

Definition 8. *We set* $\langle\!\langle ab.P \rangle\!\rangle^{\mathbf{er}} = a(x).(x \twoheadrightarrow b \mid \langle\!\langle P \rangle\!\rangle^{\mathbf{er}})$, *where* $x \twoheadrightarrow b = !x(z).\overline{b}z$ *is called a* link process. *We also define* $\langle\!\langle \rho(x).P \rangle\!\rangle^{\mathbf{er}} = \rho(x).\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$ *and similarly for delayed prefixes.* $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$ *leaves free outputs unchanged and acts homeomorphically on the other constructors.*

The transformation $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$ removes all free inputs and inserts free outputs (in the link process). We therefore expect it to return plain π processes. Moreover, the process computed in the translation of free input behaves as expected provided only the *input capability* is transmitted (the link process *at the receiver's side* exerts the input capability on x). Accordingly, we define $T_{o\mathbf{e}} = \mu X.o^{\mathbf{e}}_{\mathbf{a}}X = o^{\mathbf{e}}_{\mathbf{a}}o^{\mathbf{e}}_{\mathbf{a}}o^{\mathbf{e}}_{\mathbf{a}}\cdots$, and $T_{i\mathbf{r}} = \mu X.i^{\mathbf{r}}_{\mathbf{a}}X = i^{\mathbf{r}}_{\mathbf{a}}i^{\mathbf{r}}_{\mathbf{a}}i^{\mathbf{r}}_{\mathbf{a}}\cdots$. We let $\Gamma_{i\mathbf{r}}$ (resp. $\Gamma_{o\mathbf{e}}$) range over environments mapping all names to some $c^{\mathbf{r}}_{\mathbf{a}}T_{i\mathbf{r}}$ (resp. $c^{\mathbf{e}}_{\mathbf{a}}T_{o\mathbf{e}}$), for $c \in \{i, o, \sharp\}$.

Lemma 8 (Typing for $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$). *If* $\Gamma_{o\mathbf{e}} \vdash P$ *then* $\Gamma_{i\mathbf{r}} \vdash \langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$ *for some* $\Gamma_{i\mathbf{r}}$.

Proof. We prove by induction on P that if $\Gamma \vdash P$ then $\overline{\Gamma} \vdash \langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$. In the case for $\boldsymbol{\nu}$ we always introduce the type $\sharp^{\mathbf{r}}_{\mathbf{a}} T_{i\mathbf{r}}$. For bound prefixes we replace $c^{\mathbf{e}}_{\mathbf{a}} T_{o\mathbf{e}}$ with $\overline{c}^{\mathbf{r}}_{\mathbf{a}} T_{i\mathbf{r}}$, and for free inputs we type links with $T_{i\mathbf{r}}$ types. □

As this result shows, $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$ yields processes that only transmit the input capability. This is reminiscent of the localised π-calculus [SW01] where only the output capability is passed.

It can be noted that Lemma 8 holds because we enforce a "double contract" in the typing rules (cf. Remark 1), which allows us to typecheck bound prefixes as **e**-names (before the transformation) and as **r**-names (after).

The relationship between P and $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$ is given in terms of barbed expansion precongruence, which is a preorder in between strong and weak barbed congruence.

Definition 9 (Barbed expansion precongruence). Barbed expansion *is the largest relation* $\dot{\lesssim}$ *such that whenever* $P \dot{\lesssim} Q$,

- *if* $P \to P'$ *then* $Q \to\Rightarrow Q'$ *with* $P' \dot{\lesssim} Q'$;
- *if* $Q \to Q'$ *then* $P \to P'$ *or* $P = P'$ *with* $P' \dot{\lesssim} Q'$;
- $P \downarrow \rho$ *implies* $Q \Downarrow \rho$, *and* $Q \downarrow \rho$ *implies* $P \downarrow \rho$.

We call (resp. typed) barbed expansion precongruence ($\lesssim^c$) *the induced (resp. typed) precongruence.*

Lemma 9 (Properties of links).

1. $a : i^{\mathbf{r}}_{\mathbf{a}} T_{i\mathbf{r}}, b : o^{\mathbf{r}}_{\mathbf{a}} T_{i\mathbf{r}} \rhd a \twoheadrightarrow b \lesssim^c (\nu x)(a \twoheadrightarrow x \mid x \twoheadrightarrow b)$.
2. *If* $\Gamma_{o\mathbf{e}}, a : T_{o\mathbf{e}} \vdash P$ *then* $\Gamma_{i\mathbf{r}} \rhd \langle\!\langle P \rangle\!\rangle^{\mathbf{er}}[b/a] \lesssim^c (\nu a)(a \twoheadrightarrow b \mid \langle\!\langle P \rangle\!\rangle^{\mathbf{er}})$.

Proof. 1. The law is valid for the ordinary π-calculus (and is substitution-closed); Lemma 3 transfers the result to $\overline{\pi}$.

2. By typing, a free output involving a in $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$ is necessarily in a link; in this case, we can use (1). The other kind of interaction is with some $\overline{a}(x){:}Q$ in $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$, and $\overline{b}(x){:}Q[b/a]$ behaves like $(\boldsymbol{\nu} a)(a \twoheadrightarrow b \mid \overline{a}(x){:}Q[b/a])$. □

We use Lemma 9 to deduce operational correspondence.

Lemma 10 (Operational correspondence). *Suppose that* $\Gamma_{o\mathbf{e}} \vdash P$.

1. $P \downarrow \rho$ *iff* $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \downarrow \rho$.
2. *If* $P \to P'$ *then* $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \to\gtrsim^c \langle\!\langle P' \rangle\!\rangle^{\mathbf{er}}$.
3. *If* $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \to P_1$ *then* $P \to P'$ *and* $P_1 \gtrsim^c \langle\!\langle P' \rangle\!\rangle^{\mathbf{er}}$ *for some* P'.

A version of these results in the weak case can also be proved, for barbed expansion. Notably, P and $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}$ exhibit the same weak barbs.

Lemma 11. *If* $\Gamma_{o\mathbf{e}} \vdash P, Q$ *then* $P \dot{\approx} Q$ *iff* $\langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \dot{\approx} \langle\!\langle Q \rangle\!\rangle^{\mathbf{er}}$.

Proof. We show that $\dot{\gtrsim}\{(\langle\!\langle P \rangle\!\rangle^{\mathbf{er}}, \langle\!\langle Q \rangle\!\rangle^{\mathbf{er}}) \mid P \dot{\approx} Q\}\dot{\lesssim}$ and $\{(P, Q) \mid \langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \dot{\approx} \langle\!\langle Q \rangle\!\rangle^{\mathbf{er}}\}$ are weak barbed bisimulations. We then use the adaptation of Lemma 10 to the weak case, for barbed expansion. □

Lemma 12. *If* $\Gamma_{\mathbf{oe}} \vdash P, Q$ *and* $\Gamma_{\mathbf{ir}} \rhd \langle\!\langle P \rangle\!\rangle^{\mathbf{er}} \cong^c \langle\!\langle Q \rangle\!\rangle^{\mathbf{er}}$ *then* $\Gamma_{\mathbf{oe}} \rhd P \cong^c Q$.

Proof. We define a type system with marks on types, such that only $T_{\mathbf{ir}}$-types are marked. The marking propagates onto the names of the typed processes. We modify the encoding $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$ to only operate on marked prefixes. For every $(\Delta/\Gamma_{\mathbf{oe}})$-context C, its encoding $\langle\!\langle C \rangle\!\rangle^{\mathbf{er}}$ is a $(\Delta'/\Gamma_{\mathbf{ir}})$-context. Thanks to the compositionality of $\langle\!\langle \cdot \rangle\!\rangle^{\mathbf{er}}$, the hypothesis of the lemma implies the equivalence $\langle\!\langle C[P] \rangle\!\rangle^{\mathbf{er}} \mathrel{\dot{\approx}} \langle\!\langle C[Q] \rangle\!\rangle^{\mathbf{er}}$. We then adapt the proof of Lemma 11 to this marked encoding. □

3.3 An analysis of van Bakel and Vigliotti's encoding

As announced in Section 1, we start from an adaptation of Milner's call-by-name (cbn) encoding of [Mil92] to *strong* cbn, which also allows reductions to occur under λ. We obtain this by using a delayed prefix in the clause for λ-abstraction. The encoding, noted $[\![\cdot]\!]^{\mathcal{M}}$, is defined as follows:

$$[\![x]\!]^{\mathcal{M}}_p = \overline{x}p \qquad [\![\lambda x.M]\!]^{\mathcal{M}}_p = p(x,q).[\![M]\!]^{\mathcal{M}}_q$$
$$[\![MN]\!]^{\mathcal{M}}_p = (\nu q)\big([\![M]\!]^{\mathcal{M}}_q \mid (\nu x)(\overline{q}\langle x, p\rangle \mid !x(r).[\![N]\!]^{\mathcal{M}}_r)\big)$$

The other encoding we analyse, taken from [vBV09], is written $[\![\cdot]\!]^{\mathcal{B}}$:

$$[\![x]\!]^{\mathcal{B}}_p = x(p').p' \rightarrowtriangle p \qquad [\![\lambda x.M]\!]^{\mathcal{B}}_p = \overline{p}(x,q).[\![M]\!]^{\mathcal{B}}_q$$
$$[\![MN]\!]^{\mathcal{B}}_p = (\nu q)\big([\![M]\!]^{\mathcal{B}}_q \mid q(x,p').(p' \rightarrowtriangle p \mid !\overline{x}(r).[\![N]\!]^{\mathcal{B}}_r)\big)$$

Note that $[\![\cdot]\!]^{\mathcal{B}}$ is written in [vBV09] using asynchronous free output and restriction instead of delayed bound output. We can adopt this more concise notation since $(\nu x)(\overline{a}x \mid P)$ and $\overline{a}(x).P$ are strongly bisimilar processes, and similarly for $x(p').p' \rightarrowtriangle p$ and $x(p').p' \rightarrowtriangle p$. (Another difference is that the replication in the encoding of the application is guarded, as in [vBV10], to force a tighter operational correspondence between reductions in λ and in the encodings.)

As remarked above, $[\![\cdot]\!]^{\mathcal{B}}$ and $[\![\cdot]\!]^{\mathcal{M}}$ differ considerably because they engage in quite different dialogues with their environments: in $[\![\cdot]\!]^{\mathcal{M}}$ a function receives its argument via an input, in $[\![\cdot]\!]^{\mathcal{B}}$ it interacts via an output. Differences are also visible in the encodings of variables and application (e.g. the use of links).

To compare the encodings $[\![\cdot]\!]^{\mathcal{M}}$ and $[\![\cdot]\!]^{\mathcal{B}}$, we introduce an intermediate encoding, noted $[\![\cdot]\!]^{\mathcal{I}}$, which is defined as the dual of $[\![\cdot]\!]^{\mathcal{M}}$ (in $\overline{\pi}$):

$$[\![x]\!]^{\mathcal{I}}_p = xp \qquad [\![\lambda x.M]\!]^{\mathcal{I}}_p = \overline{p}(x,q).[\![M]\!]^{\mathcal{I}}_q$$
$$[\![MN]\!]^{\mathcal{I}}_p = (\nu q)\big([\![M]\!]^{\mathcal{I}}_q \mid (\nu x)(q\langle x, p\rangle \mid !\overline{x}(r).[\![N]\!]^{\mathcal{I}}_r)\big)$$

Note that while $[\![\cdot]\!]^{\mathcal{M}}$ and $[\![\cdot]\!]^{\mathcal{B}}$ can be expressed in π, $[\![\cdot]\!]^{\mathcal{I}}$ uses free input, and does thus not define π-calculus processes.

The three encodings given above are based on a similar usage of names. Two kinds of names are used: we refer to names that represent continuations (p, p', q, r in the encodings) as *handles*, and to names that stand for λ-calculus parameters (x, y, z) as *λ-variables*. Here is how these encodings can be typed in $\overline{\pi}$:

Lemma 13 (Typing the encodings). $[\![\cdot]\!]^{\mathcal{M}}$, $[\![\cdot]\!]^{\mathcal{B}}$ *and* $[\![\cdot]\!]^{\mathcal{I}}$ *yield processes which are typable with the respective typing environments* $\Gamma_{\mathcal{M}}, \Gamma_{\mathcal{B}}, \Gamma_{\mathcal{I}}$*, where:*

- $\Gamma_{\mathcal{M}}$ *types* λ*-variables with* $o^{\mathbf{r}}_{\mathbf{a}}H$ *and handles with* $H = \mu X.i^{\mathbf{r}}_{\mathbf{d}}\langle o^{\mathbf{r}}_{\mathbf{a}}X, X\rangle$*;*
- $\Gamma_{\mathcal{B}}$ *uses respectively* $i^{\mathbf{r}}_{\mathbf{d}}G$ *and* $o^{\mathbf{r}}_{\mathbf{a}}\langle o^{\mathbf{r}}_{\mathbf{d}}G, G\rangle$ *where* $G = \mu Y.i^{\mathbf{r}}_{\mathbf{a}}\langle o^{\mathbf{r}}_{\mathbf{d}}Y, Y\rangle$*;*
- $\Gamma_{\mathcal{I}}$ *is the dual of* $\Gamma_{\mathcal{M}}$ *(that is, it uses* $i^{\mathbf{e}}_{\mathbf{d}}\overline{H}$ *and* $\overline{H} = \mu Z.o^{\mathbf{e}}_{\mathbf{a}}\langle i^{\mathbf{e}}_{\mathbf{d}}Z, Z\rangle$*).*

Encoding $[\![\cdot]\!]^{\mathcal{I}}$ can be obtained from $[\![\cdot]\!]^{\mathcal{M}}$ by duality. The only difference between $[\![\cdot]\!]^{\mathcal{I}}$ and $[\![\cdot]\!]^{\mathcal{B}}$ is the presence of two links. We rely on a link transformation similar to the one of Section 3.2 to move from $[\![\cdot]\!]^{\mathcal{I}}$ to $[\![\cdot]\!]^{\mathcal{B}}$. Thus, by composing the results on duality and on the transformation, we are able to go from $[\![\cdot]\!]^{\mathcal{M}}$ to $[\![\cdot]\!]^{\mathcal{B}}$.

Proposition 4. *Given two* λ*-terms* M *and* N*, we have* $[\![M]\!]^{\mathcal{M}}_p \dot{\approx} [\![N]\!]^{\mathcal{M}}_p$ *if and only if* $[\![M]\!]^{\mathcal{B}}_p \dot{\approx} [\![N]\!]^{\mathcal{B}}_p$ *(both equivalences are in* $\pi^{\mathrm{io},\mathbf{a}}$*).*

Proof. By duality, $[\![M]\!]^{\mathcal{M}}_p \dot{\approx} [\![N]\!]^{\mathcal{M}}_p$ iff $[\![M]\!]^{\mathcal{I}}_p \dot{\approx} [\![N]\!]^{\mathcal{I}}_p$. To establish that this is equivalent to $[\![M]\!]^{\mathcal{B}}_p \dot{\approx} [\![N]\!]^{\mathcal{B}}_p$, we rely on an adaptation of Lemma 11. For this, we define a transformation that exploits the ideas presented in Section 3.2. In particular, handles (p, p', q, r) are treated like in Definition 8. The handling of λ-variables (x, y, z) is somehow orthogonal, and raises no major difficulty, because such names are always transmitted as bound (fresh) names. □

Remark 6 (Call by name). To forbid reductions under λ-abstractions, we could adopt Milner's original encoding, and use an input prefix instead of delayed input in the translation of abstractions. Accordingly, adapting Van Bakel and Vigliotti's encoding to this strategy would mean introducing a free input prefix — which is rather natural in $\overline{\pi}$, but is not in the π-calculus.

4 Concluding remarks

We have presented several properties of $\overline{\pi}$, and established relationships with the π-calculus with i/o-types (π^{io}).

The calculus $\overline{\pi}$ enjoys properties of dualities while being "large", in the sense that it incorporates many of the forms of prefix found in dialects of the π-calculus (free input, bound input, and, in the extension in Section 3.1, also delayed input, plus the analogue for outputs), and a non-trivial type system based on i/o-types. This syntactic abundance makes $\overline{\pi}$ a possibly interesting model in which to study various forms of dualities. This is exemplified in our study of encodings of the λ-calculus, where we have applied $\overline{\pi}$ and its theory to explain a recent encoding of cbn λ-calculus by van Bakel and Vigliotti: it can be related, via dualities, to Milner's encoding.

It would be interesting to strengthen the full abstraction in Lemma 11 from barbed bisimilarity to barbed congruence. This would allow us to replace barbed bisimilarity with typed barbed congruence in Proposition 4 as well (using the type environments of Lemma 13). While we believe the result to be true, the

proof appears difficult because the link transformation modifies both processes and types, so that the types needed for barbed congruence in the two encodings are different. Therefore also the sets of contexts to be taken into account are different. The problem could be tackled by combining the theory on delayed input and the link bisimilarity in [MS04], and adapting it to a typed setting.

We plan to further investigate the behavioural theory of $\overline{\pi}$, and study in particular other transformations along the lines of Section 3.2, where link processes are used to implement substitutions. It would be interesting to provide general results on process transformations in terms of links, when the direction and the form of the links vary depending on the types of the names involved. Currently we only know how to handle them when the calculus is asynchronous and localised [MS04].

As already mentioned, another interesting issue is how to accommodate i/o-types into πI and fusion calculi while maintaining the dualities of the untyped calculi.

Acknowledgments. This work was supported by the french ANR projects Recre, 2009-BLAN-0169-02 Panda, and 2010-BLAN-0305-01 PiCoq.

References

[Fu97] Y. Fu. The χ-calculus. In *Proc. of APDC'97*, pages 74–81. IEEE Computer Society Press, 1997.

[GW00] P. Gardner and L. Wischik. Explicit fusions. In *Proc. of MFCS*, volume 1893 of *LNCS*, pages 373–382. Springer-Verlag, 2000.

[HMS12] D. Hirschkoff, J.M. Madiot, and D. Sangiorgi. On subtyping in symmetric versions of the π-calculus. In preparation, 2012.

[Mer00] M. Merro. *Locality in the pi-calculus and applications to distributed objects.* PhD thesis, École des Mines, France, 2000.

[Mil92] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MS04] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science*, 14(5):715–767, 2004.

[PV98] J. Parrow and B. Victor. The fusion calculus: expressiveness and symmetry in mobile processes. In *Proc. of LICS*, pages 176 –185. IEEE, 1998.

[San96] D. Sangiorgi. π-calculus, internal mobility, and agent-passing calculi. In *Selected papers from TAPSOFT '95*, pages 235–274. Elsevier, 1996.

[SW01] D. Sangiorgi and D. Walker. *The Pi-Calculus: a theory of mobile processes.* Cambridge University Press, 2001.

[Vas09] V. T. Vasconcelos. Fundamentals of session types. In *Proc. of SFM*, volume 5569 of *LNCS*, pages 158–186. Springer, 2009.

[vBV09] S. van Bakel and M. G. Vigliotti. A logical interpretation of the λ-calculus into the π-calculus, preserving spine reduction and types. In *Proc. of CONCUR*, volume 5710 of *LNCS*, pages 84–98. Springer, 2009.

[vBV10] S. van Bakel and M. G. Vigliotti. Implicative logic based encoding of the λ-calculus into the π-calculus, 2010. From http://www.doc.ic.ac.uk/ svb/.