Automatically Proving Up-to Bisimulation

Daniel Hirschkoff

Mars 1998

N^o 98-123

Automatically Proving Up-to Bisimulation

Daniel Hirschkoff

CERMICS - ENPC/INRIA and Università di Roma "La Sapienza"

Résumé

Nous développons diverses méthodes permettant d'automatiser des preuves de bisimulation "up-to" pour le π-calcul. Nous traitons les techniques "à congruence structurelle près" (par l'intermédiaire de la définition d'un algorithme calculant une forme normale unique pour la congruence structurelle), "à substitutions injectives sur les noms libres près", "à restriction près", et "à composition parallèle près". Ces techniques permettent de réduire la taille des relations que l'on doit exhiber afin de prouver un résultat de bisimilarité. Dans le cadre d'une implémentation, cela présente des avantages immédiats en termes de gestion de la mémoire; en outre, sur le plan de l'expressivité de notre système, la technique de preuve "à composition parallèle près" permet de prendre en compte certains termes comportant l'opérateur de réplication, et dont l'espace d'états est infini (chose que ne peuvent faire les outils existants pour le π-calcul). Nous illustrons les divers avantages de ces méthodes sur une implémentation.

Abstract

We present a methodology for checking bisimilarities between π-calculus processes with the up-to techniques *for bisimulation. These techniques are used to reduce the size of the relation one has to exhibit to prove a bisimulation. Not only is this interesting in terms of space management, but it also increases dramatically the expressive power of our system, by making in some cases the verification of infinite states space processes possible. Based on an algorithm to compute a unique normal form for structural congruence, we develop sound and complete methods to check bisimulation up to injective substitutions on free names, up to restriction and up to parallel composition. We show the expressiveness of our techniques on a prototype implementation.*

Introduction

The π-calculus has become a widely studied model of concurrency; the most popular notion of equivalence on π-calculus terms is bisimilarity. Existing tools for automatically checking bisimilarities between CCS or π-calculus processes can handle a restricted class of processes that have an infinite behaviour. Methods like the partition refinement algorithm ([PT87], used for example in [PS96]) are based on a preliminary step in which the unfolding of the processes is computed, under the form of a *Labelled Transition System*. Therefore, processes having an infinite state space cannot be taken into consideration by this approach. The *Mobility Workbench* [VM94] uses an "on the fly" method, that progressively builds the candidate bisimulation relation as new pairs of related processes are discovered. This way, one can also take into account two non terminating processes that show a different behaviour after a finite number of steps, and prove that they are not bisimilar. However, two processes having an infinite states space still cannot be proven bisimilar.

In this paper, a different approach, based on the so called up-to proof techniques, is investigated, in order to define some methods for checking bisimilarities between processes. These techniques have been introduced as meta-level tools for proving bisimulation relations [SM92, San95], and to our knowledge have only been used in papers about the theory of π-calculus, to prove bisimilarity laws (e.g. [BS98, San96a]).

The idea behind the up-to techniques is to reduce the size of the relation one has to exhibit to establish a bisimilarity property, by providing syntactical tools to manipulate pairs of processes before checking that these pairs belong to the relation. For example, the classical proof of commutativity of the parallel composition operator exhibits a relation $\mathcal{R}$ defined as follows:

$$\mathcal{R} \;=\; \{((\nu\vec{x})\,(P|Q), (\nu\vec{x})\,(Q|P))\}\,,$$

P and Q ranging over processes and $\vec{x}$ over (possibly empty) name lists (we have to add the restriction on $\vec{x}$ because of possible name extrusions). The up to restriction proof technique makes it possible to cancel common restrictions in pairs of related processes, and thus allows us to work with the simpler relation $\mathcal{R}' \;=\; \{(P|Q, Q|P)\}$. Similarly, the up to parallel composition proof technique allows one to cut the common parallel components of a pair of processes, while the up to injective substitutions on names proof technique quotientates the relations under such substitutions.

In [San95], Sangiorgi gives a uniform presentation of the up-to techniques, and shows how they can be combined together to get powerful proof

methods. An essential characteristic of these techniques is that they are used at a purely syntactical level. This has two important consequences: on one side, our algorithms will follow a syntactical approach (as opposed to the methods where one first computes a semantical model of the processes), and the automatic proofs will be quite similar to "human" proofs, which can help tracing errors in the case where two processes are not bisimilar. On the other hand, the up-to parallel composition proof technique allows one to work with replicated processes, i.e. processes that can have an infinite behaviour. This means that in some cases we can prove bisimilarities even for infinite-states space processes, and hence treat more processes than the existing algorithms.

Since we are to work at a syntactical level, we have to be able to "deal with syntax", which in our setting can be expressed by the ability to work up to structural congruence[1]. Structural congruence (written $\equiv$) has been introduced in [Mil91, Mil92]; it is probably the most immediate syntactical equivalence that can be defined on processes, and basically says how the parallel subcomponents and the restrictions of a term can be rearranged without changing the process it actually represents. We work with a simple but expressive syntax for processes, where replications are allowed only on prefixed terms (akin to the *normalised replications* of [San95], although we do not have the sum operator in our language); within this framework, a normalising algorithm for structural congruence can be defined as a rewriting system, that guarantees uniqueness of normal form for structurally equivalent processes. This allows us to work systematically up to structural congruence, and to consider relations that contain only pairs of normal processes. We define sound and complete methods for three more up-to techniques, incrementally adding each time a new technique, namely up-to injective substitutions on free names, up-to restriction and up-to parallel composition.

The plan of the paper is as follows: in the next Section, we define the syntax of processes and structural congruence; we then define a rewriting system by orienting the structural congruence laws, obtaining a system that enjoys the strong normalisation and local confluence properties. This gives uniqueness of normal forms, for which we provide a syntactical description. We introduce semantics in Section 2, and develop the framework of the up-to bisimulation proof techniques. In Section 3, we discuss our up-to injective substitutions, up-to restriction and up-to parallel composition proof methods; Section 4 is devoted to the description of a prototype implementation

[1] The Mobility WorkBench exploits the axiomatisation of open bisimulation [San96b], since it also needs to stay at the level of syntax, because of the "on the fly" method.

of these algorithms, and illustrates the capacities of our system on several examples. We finally conclude and discuss future extensions of this work.

1 Syntax

This Section is devoted to the syntactical manipulation of processes. We first introduce the syntax of the language we study, and structural congruence; we then define a term rewriting system that computes a unique representant for every equivalence class for structural congruence, and give a syntactical characterisation of this representant.

1.1 Definitions

We let $a, b, \ldots, x, y, \ldots$ range over an infinite countable set of *names*, and $\vec{a}, \vec{b}, \ldots$ range over (possibly empty) name lists. Processes, ranged over by $P, Q, \ldots$, are defined by the following syntax:

$$\begin{array}{rcl} \alpha & = & a(\vec{b}) \mid \overline{a}[\vec{b}] \\ P & = & \mathbf{0} \mid \alpha.P \mid !\alpha.P \mid (\nu x)\, P \mid P_1|P_2\,. \end{array}$$

Prefixes (ranged over by $\alpha, \beta, \ldots$) are either input: $a(\vec{b})$, or output: $\overline{a}[\vec{b}]$. $\mathbf{0}$ is the inactive process; prefixed processes are either linear ($\alpha.P$) or replicated ($!\alpha.P$);the other constructors are restriction (ν) and parallel composition ($|$). The notion of bound name is defined by saying that restriction and input prefix are binding operators: (νx) and $x(\vec{y})$. bind respectively name x and the names in $\vec{y}$ in the process these operators are applied to. As usual, free names are names that are not bound in a process, and we work up to implicit α-conversion of bound names (at least until Section 3, where α-conversion will be handled explicitly for the definition of our checking methods).

Structural congruence, written $\equiv$, is the smallest equivalence relation that is a congruence and that satisfies the following rules:

1 $P\|\mathbf{0} \equiv P$	2 $P\|Q \equiv Q\|P$	3 $P\|(Q\|R) \equiv (P\|Q)\|R$
4 $(\nu x)(\nu y)\, P \equiv (\nu y)(\nu x)\, P$		
5 $\dfrac{x \notin fn(P)}{(\nu x)\, P \equiv P}$	6 $\dfrac{x \notin fn(P)}{P\|(\nu x)\, Q \equiv (\nu x)\, (P\|Q)}$	
7 $!\alpha.P\|\alpha.P \equiv !\alpha.P$	8 $!\alpha.P\|!\alpha.P \equiv !\alpha.P$	

Rules 1-3 give properties about the parallel composition operator, rules 4-6 deal with restriction, and rules 7-8 with replication. Rule 8 is new with respect to the traditional definition of structural congruence [Mil91], and can be seen as the "limit" of infinitely many applications of Rule 7.

Conventions and notations: Rules 2 and 3 (for parallel composition) and 4 (for restriction) will be used implicitly, which means that we work up to commutativity and associativity of parallel composition, and up to permutation of consecutive restrictions. This will allow us to use the notation $(P_1|\ldots|P_n)$ (sometimes abbreviated as $\prod_{i\in[1,\ldots,n]} P_i$) for parallel composition, and $(\nu\vec{x})\,P$ for restriction, where $\vec{x}$ is intended as having a set rather than a vector structure (in order to allow silent applications of rule 4).

Whereas rules 1 and 5 are used to do some garbage collection; the rules that will be really relevant in the definition of our notion of normal form are rules 6, 7 and 8, as will be seen below. Remark that structural congruence preserves free names, i.e if $P \equiv Q$, then $fn(P) = fn(Q)$.

1.2 Normal Forms

In order to quotientate our syntax with respect to structural congruence, we define a normalisation algorithm as a term rewriting system, by orienting the laws of structural congruence (except laws 2, 3 and 4, that are used implicitly: this amounts to suppose that we have some way to sort names and parallel compositions of processes, typically using a lexicographical order; we will come back to this point in Section 3). All these laws except one relate two processes that do not have the same number of symbols, so it is natural to orientate them from the "verbose form" to the more compact one, with the aim of choosing the more succinct representation for normal forms. Only law 6, that is used to move around restrictions among parallel compositions, does not satisfy this property. Remember that it writes:

$$6 \quad \frac{x \notin fn(P)}{P|(\nu x)\,Q \equiv (\nu x)\,(P|Q)}\,.$$

It seems sensible to decide that the "right" place for the restriction on x is as close as possible to its actual scope, which would mean choosing a right-to-left orientation for the law. Unfortunately, this is not compatible with our will to have a unique normal form for two structurally congruent processes; consider for example three processes P, Q and R, and two names a and b such that a occurs free in P and Q but not in R, and b occurs free in Q and R but not in P. Then one could rewrite $(\nu a)(\nu b)\,(P|Q|R)$ in two different ways, obtaining $(\nu a)\,(P|(\nu b)\,(Q|R))$ and $(\nu b)\,((\nu a)\,(P|Q)|R)$, that

would be different normal forms if P, Q and R cannot be rewritten[2]. We are therefore compelled to orientate law 6 from left to right, hence pulling up all restrictions as much as possible in a term.

Definition 1.1 (Normalisation algorithm) *The normalisation algorithm is defined as the rewriting system given by the five following rules:*

$$R1 \quad P|\mathbf{0} \to P \qquad R5 \quad (x \notin fn(P)) \Rightarrow ((\nu x)\, P \to P)$$

$$R6 \quad (x \notin fn(P)) \Rightarrow (P|(\nu x)\, Q \to (\nu x)\, (P|Q))$$

$$R7 \quad !\alpha.P|\alpha.P \to !\alpha.P \qquad R8 \quad !\alpha.P|!\alpha.P \to !\alpha.P$$

(we have kept the numbering of the rules consistent with the structural laws)[3]. This rewriting system enjoys strong normalisation (a computation always terminates) and local confluence (two one-step reducts of a term can always be rewritten into a common term), which guarantees uniqueness of normal forms:

Lemma 1.2 (Strong normalisation) *Our rewriting system is strongly normalising.*

Proof. We want to define a measure on processes as a natural number that strictly decreases each time a rewriting rule is triggered. As remarked above, all rules except R6 make the number of symbols in the term decrease, while an application of rule R6 make the number of symbols in the scope of the restriction grow. We therefore define the measure $m(P)$, given a process P, as

$$m(P) = (\#^{\nu}(P) + 1) * \#^{symb}(P) - \sum_{\nu \text{ in } P} \#^{symb}(scope(\nu))\,,$$

where $\#^{symb}(X)$ denotes the number of symbols in X, $\#^{\nu}$ the number of restrictions, and given a term of the form $(\nu x)\, T$, the scope of the restriction on x, written $scope(\nu)$, is T.

[2] A similar problem appears in [EG96], where normal forms do not enjoy the uniqueness property; however, this is not problematic in the framework of Engelfriet and Gelsema because their multiset semantics makes the locations of restrictions disappear, which amounts to quotientate with respect to law 6.

[3] Note that rules R5 and R6 are guarded by a condition; however, we can consider our system as a usual Term Rewriting System (i.e. without conditions), for example by adopting a De Bruijn notation for names, which intrinsically embeds the side conditions when applying the structural congruence rules.

First note that $m(P)$ is always positive. Each time a rewrite law different from R6 is applied, the number of symbols in P decreases, and so does $m(P)$ (the positive part decreases strictly more than the negative part increases). When rule R6 is triggered, the positive part does not change, and the negative part increases, which makes $m(P)$ decrease. □

Lemma 1.3 (Local confluence) *Our rewriting system is locally confluent.*

Proof. An easy inspection of the confluence for the critical pairs induced by our rewriting rules. □

Notation: In the following, we write $=_{\alpha\nu}$ to denote equality between processes up to α-conversion, permutation of consecutive restrictions (structural congruence law 4), and commutativity and associativity of parallel composition. With this notation, we focus on the interplay between α-conversion and permutation of consecutive restrictions, leaving the management of parallel compositions aside, as this is a somewhat orthogonal question. We will return to this point in Section 3.

We now come to the main property of our system:

Proposition 1.4 (Uniqueness of normal forms) *For any process P, there exists a unique process, written $\mathsf{NR}(P)$, obtained by application of our rewriting system to P , and that cannot be further rewritten. Moreover, given two processes P and Q, $P \equiv Q$ if and only if $\mathsf{NR}(P) =_{\alpha\nu} \mathsf{NR}(Q)$.*

Proof. The fact that $\mathsf{NR}(P)$ is well defined and unique follows from the two Lemmas above, using Newmann's Lemma. It is obvious that the rewriting system preserves structural congruence, hence the $\Leftarrow$ direction. The $\Rightarrow$ direction follows from the uniqueness property. □

The normal form is the most "compact" representation of an equivalence class for structural congruence, in that it minimises the number of symbols in the term (see the proof of Lemma 1.2). Since our rewriting system is strongly normalising and enjoys the Church-Rosser property, we can apply whatever strategy we want to rewrite a process, yielding the following easy result:

Lemma 1.5 *For any process P and name list $\vec{x}$, $\mathsf{NR}((\nu\vec{x})\,P) = (\nu\vec{x}_{|fn(P)})\,\mathsf{NR}(P)$, where $\vec{x}_{|fn(P)}$ denotes the list of names in $\vec{x}$ that occur free in P.*

For $m \in \mathcal{N} \cup \{\omega\}$, define $(\alpha.N)^m = \overbrace{\alpha.N|\ldots|\alpha.N}^{m \text{ times}}$ if $m \in \mathcal{N}$ and $(\alpha.N)^\omega = !\alpha.N$.

$$
\begin{array}{rcl}
N & = & \mathbf{0} \\
& | & (\nu \vec{x})\,((\alpha_1.N_1)^{m_1}|\ldots|(\alpha_n.N_n)^{m_n})\,, \qquad n \geq 1,\ m_i \in \mathcal{N} \cup \{\omega\} \\
& & \qquad \left\{ \begin{array}{l} \forall i.\, x_i \in fn((\alpha_1.N_1)^{m_1}|\ldots|(\alpha_n.N_n)^{m_n}) \\ \forall i,j \in [1,\ldots,n].\, (i \neq j) \Rightarrow (\alpha_i.N_i \neq \alpha_j.N_j) \end{array} \right.
\end{array}
$$

Figure 1: Syntax of normal forms

Proposition 1.6 (Syntactical description of normal forms) *The terms that are of the form $\mathsf{NR}(P)$, for some P, are exactly those described by the syntax defined in Figure 1.*

Proof. A term of the form given in Figure 1 cannot be rewritten. Reciprocally, we prove by induction over the structure of a process P that if P cannot be further reduced, then P obeys the syntax above. If $P = \mathbf{0}$, the result is immediate. If $P = (\nu a)\,P'$, then either P' can be reduced, in which case P can also be reduced, and we get a contradiction, or P' cannot be further reduced, hence P' is described by the syntax above. The case $P' = \mathbf{0}$ is impossible, since one could reduce P; hence P' is of the form $(\nu \vec{x})\ \prod_i (\alpha_i.N_i)^{m_i}$, and $a \in fn(P')$, which shows that P is described by the syntax above. If $P = \alpha.P'$, then if P' cannot be reduced, P' is described by the syntax above, which implies that $\alpha.P$ is also of that form (the same holds for $P = !\alpha.P'$). Finally, if $P = P_1|\ldots|P_n$, then none of the P_is can be reduced; we easily show that every P_i is of the form $(\nu \vec{x_i})\ \prod_j (\alpha_j^i.N_j^i)^{m_j^i}$, with $\vec{x_i} = \emptyset$, since otherwise one could reduce P either using rule R1 or rule R6. Then, since P cannot be rewritten, $\prod_i \prod_j (\alpha_j^i.N_j^i)^{m_j^i}$ satisfies the condition given in the last line of Figure 1, and P is described by the syntax above. □

Example: $\mathsf{NR}((\nu a)\,(!ax.\overline{x} \mid !\overline{a}b) \mid (\nu a)\,(!ax.\overline{x} \mid !\overline{a}c))$
$= (\nu a)(\nu a')\,(!ax.\overline{x} \mid !a'x.\overline{x} \mid !\overline{a}b \mid !\overline{a}c)\,.$

Let us comment on the shape of normal forms. The syntax of Figure 1 says that every process that is not equivalent to $\mathbf{0}$ can be viewed as an agent with two components, its body and topmost restrictions. The body

is a parallel composition of prefixed processes, that, as will be seen in the next Section, are the processes that are ready to commit. The topmost restrictions define some kind of geometry among the prefixed processes, by making some names private to subsets of the body.

Notations: Given a non-null process in normal form $P = \prod_i (\alpha_i.N_i)^{m_i}$, we write $P = (\nu \vec{x}_P)\,|P|$ to decompose P into its uppermost restrictions $(\nu \vec{x}_P)$ and its "body" $\prod_i (\alpha_i.N_i)^{m_i}$, which consists of (possibly replicated) prefixed processes. Note that bodies of normal forms are also normal forms. We will range over such processes (i.e. non-null normal forms without topmost restrictions) with the notation $|P|, |Q|, \ldots$. We further decompose $|P|$ into an "infinite part", written $|P|_\omega$, and a "finite part", written $|P|_{\mathcal{N}}$, respectively corresponding to the replicated and the non-replicated components, i.e. $|P|_\omega = \prod_{i.\, m_i = \omega} (\alpha_i.N_i)^{m_i}$ and $|P|_{\mathcal{N}} = \prod_{i.\, m_i \in \mathcal{N}} (\alpha_i.N_i)^{m_i}$.

We introduce some machinery on processes of the form $|P|$, in order to establish a result that will be useful for the treatment of the up to parallel composition proof technique in Section 3: we let

$$\prod_i (\alpha_i.P_i)^{m_i} \setminus \prod_j (\beta_j.Q_j)^{\omega} \stackrel{def}{=} \prod_{i.\forall j.\, \alpha_i.P_i \neq \beta_j.Q_j} (\alpha_i.P_i)^{m_i}$$

(note that the right hand side argument of $\setminus$ is always of the form $|P|_\omega$), and, for two processes of the form $|P|$ and $|Q|$, we let

$$|P| \oplus |Q| \stackrel{def}{=} |P|_\omega \mid (|Q|_\omega \setminus |P|_\omega) \mid (|P|_{\mathcal{N}} \setminus |Q|_\omega) \mid (|Q|_{\mathcal{N}} \setminus |P|_\omega)\,.$$

Operator $\oplus$ captures the transformation that is done in the normalisation of a parallel composition of normal forms:

Lemma 1.7 $\mathsf{NR}(|P| \mid |Q|) = |P| \oplus |Q|$.

Corollary 1.8 *Let $(\nu \vec{x})\,|P|$ and $(\nu \vec{y})\,|Q|$ be two processes in normal form. Then*

$$\mathsf{NR}((\nu \vec{x})\,|P| \mid (\nu \vec{y})\,|Q|) = (\nu \vec{x}\vec{y})\,(|P| \oplus |Q|)\,.$$

2 Semantics

We now introduce the semantical notions we need in order to reason about processes, i.e. the labelled transition system and the behavioural equivalence that we will use in the remainder of the paper, namely bisimilarity.

2.1 Operational Semantics and Bisimulation

We introduce a labelled transition system by defining a judgement of the form $P \stackrel{\mu}{\rightarrow} P'$, meaning that process P is liable to perform action μ and then

behave like process P'. Actions, ranged over by μ, are of three kinds: *inputs* $a(\vec{b})$, *bound outputs* $(\nu\vec{b'})\,\overline{a}[\vec{b}]$ (with $\vec{b'} \subseteq \vec{b}$ - when $\vec{b'} = \emptyset$ the output is *free*), and τ, denoting internal communication. Bound names in actions occur only in bound output: the names in $\vec{b'}$ are bound in $(\nu\vec{b'})\,\overline{a}[\vec{b}]$. The rules for *early* transition semantics are given in Figure 2 (:: denotes the adjunction of an element to a list; symmetrical versions of rules PAR_l and $CLOSE_1$ have been omitted; note the particular shape of rule $BANG$, in relation to our syntax).

$$\text{INP}\quad a(\vec{x}).P \xrightarrow{a(\vec{b})} P_{\{\vec{x}:=\vec{b}\}} \qquad \text{RES}\quad \frac{P \xrightarrow{\mu} P'}{(\nu x)\,P \xrightarrow{\mu} (\nu x)\,P'} \quad x \notin n(\mu)$$

$$\text{OUT}\quad \overline{a}[\vec{b}].P \xrightarrow{\overline{a}[\vec{b}]} P \qquad \text{OPEN}\quad \frac{P \xrightarrow{(\nu\vec{b'})\,\overline{a}[\vec{b}]} P'}{(\nu x)\,P \xrightarrow{(\nu t::\vec{b'})\,\overline{a}[\vec{b}]} P'} \quad \begin{cases} x \neq a \\ x \in \vec{b} \setminus \vec{b'} \end{cases}$$

$$\text{BANG}\quad \frac{\alpha.P \xrightarrow{\mu} P'}{!\alpha.P \xrightarrow{\mu} !\alpha.P|P'} \qquad \text{PAR}_l\quad \frac{P \xrightarrow{\mu} P'}{Q|P \xrightarrow{\mu} Q|P'} \quad fn(Q) \cap bn(\mu) = \emptyset$$

$$\text{CLOSE}_1\quad \frac{P \xrightarrow{a(\vec{b})} P' \qquad Q \xrightarrow{(\nu\vec{b'})\,\overline{a}[\vec{b}]} Q'}{P|Q \xrightarrow{\tau} (\nu\vec{b'})\,(P'|Q')}$$

Figure 2: Early Transition Semantics

The derivations one can construct for normal forms are related to the form of processes. Since all restrictions are brought on top, in some sense name extrusion is performed "before" the transition, which implies that rule CLOSE is always used with an empty list $\vec{b'}$ of names, and that the side condition for rule PAR is always verified (since necessary α-conversion has already been done as rewrite rule R6 is applied). These considerations are taken into account for the definition of the function that computes the derivations of a normal form in our implementation (see Section 4).

The semantical equivalence on processes we use is bisimilarity, defined as follows:

Definition 2.1 (Bisimulation, bisimilarity) *A relation $\mathcal{R}$ is a* bisimulation *iff for every pair of processes (P, Q) such that $P\mathcal{R}Q$, whenever $P \xrightarrow{\mu} P'$, there exists a process Q' such that $Q \xrightarrow{\mu} Q'$ and $P'\mathcal{R}Q'$, and the symmetrical condition on transitions performed by Q. Bisimilarity, written $\sim$, is the greatest bisimulation.*

Remark: bisimilarity contains structural congruence, i.e. $\equiv \subseteq \sim$.

2.2 The Up-to Proof Techniques for Bisimulation

To rephrase Definition 2.1 above, proving bisimilarity of two processes reduces to exhibiting a bisimulation relation that contains these processes. The property "to be a bisimulation relation" can be depicted by the diagram on the left side of Figure 3: $\mathcal{R}$ is a bisimulation if any pair of processes in $\mathcal{R}$ evolves to pairs of processes that are also in $\mathcal{R}$. In other words, $\mathcal{R}$ contains the whole "future" of all the processes it relates.

$$\begin{array}{ccc} P & \mathcal{R} & Q \\ \downarrow \mu & & \downarrow \mu \\ P' & \mathcal{R} & Q' \end{array} \qquad\qquad \begin{array}{ccc} P & \mathcal{R} & Q \\ \downarrow \mu & & \downarrow \mu \\ P' & \mathcal{F}(\mathcal{R}) & Q' \end{array}$$

Figure 3: From bisimulation to up-to bisimulation

In [San95], Sangiorgi introduces a general framework for the study of the *up-to techniques*, which can be used to reduce the size of the relations one has to exhibit in order to prove bisimulation. Each such a technique is represented by a functional from relations to relations (ranged over by $\mathcal{F}$); the property of being a bisimulation *up-to* $\mathcal{F}$ is defined as follows:

Definition 2.2 (Bisimulation up to $\mathcal{F}$) *Given a functional $\mathcal{F}$ over relations, we say that a relation $\mathcal{R}$ is a bisimulation up to $\mathcal{F}$ iff, for every P and Q such that $P\mathcal{R}Q$, whenever $P \xrightarrow{\mu} P'$, there exists Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\mathcal{F}(\mathcal{R})Q'$, and the symmetrical condition on transitions performed by Q.*

A functional $\mathcal{F}$ is *sound* if ($\mathcal{R}$ is a bisimulation up to $\mathcal{F}$) implies ($\mathcal{R} \subseteq \sim$), i.e. in some way, $\mathcal{F}$ helps building the "future" of a relation: to prove that $\mathcal{R}$ is a bisimulation relation, it is enough to prove that any pair of processes in $\mathcal{R}$ can only evolve to pairs of processes that are contained in $\mathcal{F}(\mathcal{R})$ (as shown on the right part of Figure 3). [San95] introduces a sufficient condition for soundness of functionals, called *respectfulness*. All the techniques we are using in the remainder of the paper (up to injective substitutions on free names, up to structural congruence, up to restrictions, up to parallel composition) are respectful, and hence induce correct proof techniques for

bisimulation. Moreover, respectful functions enjoy nice compositional properties, and can indeed be viewed as tools to compute the closure of a relation, that can be combined together for the task of proving bisimulations.

3 Automatising the Up-to Techniques

In this Section, we define sound and complete methods to decide, given a relation $\mathcal{R}$, if a pair of processes belongs to $\mathcal{F}(\mathcal{R})$, for some function $\mathcal{F}$ corresponding to a correct proof technique. As hinted before, we systematically use the up to structural congruence proof technique, which amounts to work only with processes in normal form; this technique will be first used in conjunction with the up to injective substitutions proof method, and then we will incrementally add the up to restrictions and the up to parallel composition proof techniques, yielding our most powerful tool for bisimulation proofs. To work with normal forms will allow us to decompose any process into its body and topmost restrictions, which, as will be seen, will be our "canonical way" to reason about processes.

3.1 Up to Injective Substitutions on Free Names

As names are the only basic entity on which processes are defined, the definition of up-to techniques to work with them will be crucial in bisimulation proofs. Two methods are directly related to the management of names, namely the up-to injective substitutions on free names and the up-to restriction proof techniques. As will be seen below, their treatment will be quite uniform, the up-to restrictions checking method being in some way a generalisation of the up-to injective substitutions on free names checking method, which in turn can also be seen as an extension of the α-conversion checking method. To this end, we shall work with the decomposition of normal processes into their topmost restrictions and bodies, and study the application of (injective) substitutions to the bodies. Let us first introduce some background on substitutions.

Substitutions: We work with substitutions on names, that are functions from names to names, ranged over by σ, σ', σ''. We define $dom(\sigma)$, the domain of σ, as the set of names n such that $\sigma(n) \neq n$, and the codomain $codom(\sigma)$ of σ as $\sigma(dom(\sigma))$. σ is *injective* if $\sigma(i) = \sigma(j)$ implies $i = j$. In the following, we are interested, given a process P, in substitutions σ that are injective on the free names of P, and such that applying σ to P does not capture bound names of P, i.e. $codom(\sigma) \cap bn(P) = \emptyset$ (this is always possible modulo α-conversion). An injective substitution σ whose domain is

finite defines a bijective mapping between $dom(\sigma)$ and $codom(\sigma)$; we shall write σ^{-1} for the inverse of σ. Given a set of names E, we say that two substitutions σ and σ' *coincide* on E iff for any name n in E, $\sigma(n) = \sigma'(n)$.

We can remark that since NR preserves free names, we have, for a process P and a substitution σ injective on the free names of P, $\mathsf{NR}(P\sigma) = \mathsf{NR}(P)\sigma$.

We are now ready to define our proof techniques using injective substitutions on free names. Let us start with an obvious characterisation of α-convertibility:

Fact 3.1 *Given two processes P and Q in normal form, write $P = (\nu\vec{x})\,|P|$ and $Q = (\nu\vec{y})\,|Q|$. Then $(\nu\vec{x})\,|P| =_{\alpha\nu} (\nu\vec{y})\,|Q|$ iff there exists a substitution σ, injective on $fn(|Q|)$, s.t. $|P| =_{\alpha\nu} |Q|\sigma$, $dom(\sigma) = \vec{y}$ and $\sigma(\vec{y}) = \vec{x}$.*

A direct consequence of the above result:

Lemma 3.2 *Write as above $P = (\nu\vec{x})\,|P|$ and $Q = (\nu\vec{y})\,|Q|$, for two normal processes P and Q. Then, for any substitution σ injective on the free names of $|Q|$, $(\nu\vec{x})\,|P| =_{\alpha\nu} (\nu\vec{y})\,(|Q|\sigma)$ iff there exists a substitution σ' injective on $fn(|Q|)$ s.t. (i) $|P| =_{\alpha\nu} |Q|\sigma'$, (ii) $\sigma'(\sigma^{-1}(\vec{y})) = \vec{x}$, and (iii) $\sigma' = \sigma$ on $fn(|Q|) \setminus \sigma^{-1}(\vec{y})$.*

We can now consider the closure under structural congruence and injective substitutions on free names of a relation:

Definition 3.3 *Given a relation $\mathcal{R}$, we define the* closure under structural congruence and injective substitutions on free names *of $\mathcal{R}$, written $\equiv\mathcal{R}^i\equiv$, as follows:*

$$\equiv\mathcal{R}^i\equiv \stackrel{def}{=} \{(P,Q); \exists (P_0,Q_0) \in \mathcal{R}, \exists \sigma \text{ injective on } fn(P_0)\cup fn(Q_0) \text{ s.t. } P \equiv P_0\sigma \wedge Q \equiv Q_0\sigma\}.$$

Proposition 3.4 (Characterisation of $\equiv\mathcal{R}^i\equiv$) *$(P,Q) \in \equiv\mathcal{R}^i\equiv$ iff there exist processes P_0, Q_0, and substitutions σ' σ'', injective on $fn(|P_0|)$ and $fn(|Q_0|)$ respectively, such that, if we write $P_0 = (\nu\vec{x}_{P_0})\,|P_0|$, $Q_0 = (\nu\vec{x}_{Q_0})\,|Q_0|$, $\mathsf{NR}(P) = (\nu\vec{x}_P)\,|\mathsf{NR}(P)|$ and $\mathsf{NR}(Q) = (\nu\vec{x}_Q)\,|\mathsf{NR}(Q)|$, we have $P_0\mathcal{R}Q_0$, $|\mathsf{NR}(P)| =_{\alpha\nu} |P_0|\sigma'$, $|\mathsf{NR}(Q)| =_{\alpha\nu} |Q_0|\sigma''$, $\sigma'(\vec{x}_{P_0}) = \vec{x}_P$, $\sigma''(\vec{x}_{Q_0}) = \vec{x}_Q$, and, if we write $E = fn(|P_0|) \cap fn(|Q_0|) \setminus \vec{x}_{P_0}\vec{x}_{Q_0}$, $\sigma' = \sigma''$ on E.*

Proof. By definition, $(P,Q) \in \equiv\mathcal{R}^i\equiv$ iff there exist $(P_0,Q_0) \in \mathcal{R}$ and σ injective s.t. $P \equiv P_0\sigma$ and $Q \equiv Q_0\sigma$, which is equivalent, using Proposition

1.4 and decomposing the normal forms into their topmost restrictions and bodies, to $(\nu\vec{x}_P)\,|\mathsf{NR}(P)|{=}_{\alpha\nu}((\nu\vec{x}_{P_0})\,|P_0|)\sigma$ and $(\nu\vec{x}_Q)\,|\mathsf{NR}(Q)|{=}_{\alpha\nu}((\nu\vec{x}_{Q_0})\,|Q_0|)\sigma$. σ being defined on the free names of P_0 and Q_0, $dom(\sigma)\cap\vec{x}_{P_0} = dom(\sigma)\cap\vec{x}_{Q_0} = \emptyset$, hence $\sigma(\vec{x}_{P_0}) = \vec{x}_{P_0}$ and $\sigma(\vec{x}_{Q_0}) = \vec{x}_{Q_0}$. Furthermore, we can suppose that sets $fn(P_0)$, $fn(Q_0)$, $\vec{x}_{P_0}$ and $\vec{x}_{Q_0}$ are pairwise distinct. We thus have: $(P,Q)\in\equiv\mathcal{R}^i\equiv$ iff there exist $(P_0,Q_0)\in\mathcal{R}$ and an injective substitution σ s.t.

$$\left\{\begin{array}{l}\sigma(\vec{x}_{P_0}) = \vec{x}_{P_0}\\ \sigma(\vec{x}_{Q_0}) = \vec{x}_{Q_0}\end{array}\right. \text{ and } \left\{\begin{array}{l}(\nu\vec{x}_P)\,|\mathsf{NR}(P)|{=}_{\alpha\nu}(\nu\vec{x}_{P_0})\,(|P_0|\sigma)\\ (\nu\vec{x}_Q)\,|\mathsf{NR}(Q)|{=}_{\alpha\nu}(\nu\vec{x}_{Q_0})\,(|Q_0|\sigma)\end{array}\right. .$$

We now prove the equivalence between the latter statement and the characterisation given in the Proposition.

$\underline{\Rightarrow}$: Using Lemma 3.2, we can exhibit a substitution σ', injective on $fn(|P_0|)$, and s.t. $|\mathsf{NR}(P)| = |P_0|\sigma'$, $\sigma'(\sigma^{-1}(\vec{x}_{P_0})) = \vec{x}_P$ and $\sigma' = \sigma$ on $fn(|P_0|)\setminus\sigma^{-1}(\vec{x}_{P_0})$. But we know from above that $\sigma^{-1}(\vec{x}_{P_0}) = \sigma(\vec{x}_{P_0}) = \vec{x}_{P_0}$, and hence we have $\sigma'(\vec{x}_{P_0}) = \vec{x}_P$, and, since $fn(|P_0|)\setminus\vec{x}_{P_0}\subseteq E = fn(|P_0|)\cap fn(|Q_0|)\setminus\vec{x}_{P_0}\vec{x}_{Q_0}$, we also have $\sigma' = \sigma$ on E. We proceed similarly for Q and Q_0, exhibiting σ'' s.t. $|\mathsf{NR}(Q)| = |Q_0|\sigma''$, $\sigma''(\vec{x}_{Q_0}) = \vec{x}_Q$ and $\sigma'' = \sigma$ on E (hence σ, σ' and σ'' coincide on E).

$\underline{\Leftarrow}$: write $fn(|P_0|)\cup fn(|Q_0|)$ as the disjoint union:

$$\vec{x}_{P_0}\cup\vec{x}_{Q_0}\cup(fn(|P_0|)\setminus(fn(|Q_0|)\cup\vec{x}_{P_0}))\cup(fn(|Q_0|)\setminus(fn(|P_0|)\cup\vec{x}_{Q_0}))\cup E\,.$$

We define a substitution σ over this set as follows:

$$\begin{array}{ll}\sigma = \sigma'\ (=\sigma'') \text{ on } E; & \sigma = \sigma' \text{ on } fn(|P_0|)\setminus(\vec{x}_{P_0}\cup fn(|Q_0|));\\ \sigma(\vec{x}_{P_0}) = \vec{x}_{P_0} \text{ and } \sigma(\vec{x}_{Q_0}) = \vec{x}_{Q_0}; & \sigma = \sigma'' \text{ on } fn(|Q_0|)\setminus(\vec{x}_{Q_0}\cup fn(|P_0|)).\end{array}$$

By definition, σ is injective on $fn(|P_0|)\cup fn(|Q_0|)$, σ and σ' coincide on $fn(|P_0|)\setminus\vec{x}_{P_0} = (fn(|P_0|)\setminus(\vec{x}_{P_0}\cup fn(|Q_0|)))\cup E$, and since $\sigma(\vec{x}_{P_0}) = \vec{x}_{P_0}$, $\sigma'(\sigma^{-1}(\vec{x}_{P_0})) = \sigma'(\vec{x}_{P_0}) = \vec{x}_P$. This allows us to show, using Lemma 3.2, that $(\nu\vec{x}_P)\,|\mathsf{NR}(P)|{=}_{\alpha\nu}(\nu\vec{x}_{P_0})\,(|P_0|\sigma)$. We prove similarly that $(\nu\vec{x}_Q)\,|\mathsf{NR}(Q)|{=}_{\alpha\nu}(\nu\vec{x}_{Q_0})(|Q_0|\sigma)$, which concludes the proof. □

This characterisation of the up to structural congruence, up to injective substitutions closure gives us a method to check that given P, Q and $\mathcal{R}$, we have $(P,Q)\in\equiv\mathcal{R}^i\equiv$: compute $\mathsf{NR}(P)$ and $\mathsf{NR}(Q)$, and, for $(P_0,Q_0)\in\mathcal{R}$, compute the substitutions σ' and σ'', and check for the conditions of Proposition 3.4; if the conditions are satisfied, succeed, otherwise choose another pair in $\mathcal{R}$.

3.2 Up to Restriction

Definition 3.5 *We write $\equiv(\mathcal{R}^i)^\nu\equiv$ to denote the closure under injective substitutions, structural congruence and restrictions of a relation $\mathcal{R}$, defined as follows:*

$$\equiv(\mathcal{R}^i)^\nu\equiv \stackrel{def}{=} \{(P,Q);\exists P_0,Q_0,\exists\vec{v},\exists\sigma \text{ injective on } fn(P_0).\\ ((P_0,Q_0)\in\mathcal{R})\wedge(P\equiv(\nu\vec{v})\,(P_0\sigma))\wedge(Q\equiv(\nu\vec{v})\,(Q_0\sigma))\}\,.$$

Proposition 3.6 (Characterisation of the closure up to restrictions) *Given two processes P and Q and a relation $\mathcal{R}$, write $\mathbf{NR}(P)=(\nu\vec{x}_P)\,|\mathbf{NR}(P)|$ and $\mathbf{NR}(Q)=(\nu\vec{x}_Q)\,|\mathbf{NR}(Q)|$. Then $(P,Q)\in\equiv(\mathcal{R}^i)^\nu\equiv$ iff there exist two substitutions σ' and σ'' injective on $fn(|P_0|)$ and $fn(|Q_0|)$ respectively, processes P_0 and Q_0 and a name list $\vec{V}\subseteq fn(P_0)\cup fn(Q_0)$ such that, if we write $P_0=(\nu\vec{x}_{P_0})\,|P_0|$, $Q_0=(\nu\vec{x}_{Q_0})\,|Q_0|$, $\vec{V}_1=\vec{V}_{|fn(P_0))}$ and $\vec{V}_2=\vec{V}_{|fn(Q_0)}$:*

$$(i)\ |\mathbf{NR}(P)|=_{\alpha\nu}|P_0|\sigma' \quad and \quad |\mathbf{NR}(Q)|=_{\alpha\nu}|Q_0|\sigma''$$
$$(ii)\ \sigma'(\vec{V}_1\vec{x}_{P_0})=\vec{x}_P \quad and \quad \sigma''(\vec{V}_2\vec{x}_{Q_0})=\vec{x}_Q$$
$$(iii)\ \sigma'=\sigma'' \text{ on } E=fn(|P_0|)\cap fn(|Q_0|)\setminus(\vec{V}\vec{x}_{P_0}\vec{x}_{Q_0})\,.$$

Proof. The proof is quite similar to the proof of Proposition 3.4. First remark that in the definition of the closure under restrictions of a relation, we can restrict ourselves to name lists $\vec{v}$ that are included in $fn(P_0)\cup fn(Q_0)$, since for every $x\notin fn(P_0)\cup fn(Q_0)$, $\vec{v}$ fits to the definition of the closure under restrictions if and only if $(x::\vec{v})$ fits.

As above, we first give an intermediate characterisation of $\equiv(\mathcal{R}^i)^\nu\equiv$, and then prove its equivalence with the conditions of the statement of the Proposition. By applying Proposition 1.4, we compute the normal forms of P and Q, which are then simplified using Lemma 1.5. Supposing as above that $fn(P_0)$, $\vec{x}_{P_0}$, $fn(Q_0)$ and $\vec{x}_{Q_0}$ are pairwise disjoint sets, we finally have the following characterisation: $(P,Q)\in\equiv(\mathcal{R}^i)^\nu\equiv$ iff there exist $(P_0,Q_0)\in\mathcal{R}$, a substitution σ injective on $fn(|P_0|)\cup fn(|Q_0|)$, and a name list $\vec{v}\subseteq\sigma(fn(P_0)\cup fn(Q_0))$ s.t.

$$\begin{cases}\sigma(\vec{x}_{P_0})=\vec{x}_{P_0}\\ \sigma(\vec{x}_{Q_0})=\vec{x}_{Q_0}\end{cases}\quad\text{and}\quad\begin{cases}(\nu\vec{x}_P)\,|\mathsf{NR}(P)|=_{\alpha\nu}(\nu\vec{v}_{|\sigma(fn(P_0))})\,(\nu\vec{x}_{P_0})\,(|P_0|\sigma)\\ (\nu\vec{x}_Q)\,|\mathsf{NR}(Q)|=_{\alpha\nu}(\nu\vec{v}_{|\sigma(fn(Q_0))})\,(\nu\vec{x}_{Q_0})\,(|Q_0|\sigma)\end{cases}.$$

$\underline{\Rightarrow:}$ write $\vec{v}_1=\vec{v}_{|\sigma(fn(P_0))}$ and $\vec{v}_2=\vec{v}_{|\sigma(fn(Q_0))}$. Lemma 3.2 gives us a substitution σ' s.t. $|\mathsf{NR}(P)|=_{\alpha\nu}|P_0|\sigma'$, $\sigma'(\sigma^{-1}(\vec{v}_1\vec{x}_{P_0}))=\vec{x}_P$, and $\sigma'=\sigma$ on $fn(|P_0|)\setminus\sigma^{-1}(\vec{v}_1\vec{x}_{P_0})$. Since σ is injective, $\sigma^{-1}(\vec{v}_1\vec{x}_{P_0})=\sigma^{-1}(\vec{v}_1)\sigma^{-1}(\vec{x}_{P_0})=\sigma^{-1}(\vec{v}_1)\vec{x}_{P_0}$. Write $\vec{V}_1=\sigma^{-1}(\vec{v}_1)$, then $\sigma(\vec{V}_1\vec{x}_{P_0})=\vec{x}_P$, and σ and σ' coincide

on $fn(|P_0|) \setminus \vec{V_1}\vec{x}_{P_0}$. Similarly, we write $\vec{V_2} = \sigma^{-1}(\vec{v_2})$, and we can exhibit σ'' such that $|\mathsf{NR}(Q)|=_{\alpha\nu}|Q_0|\sigma''$, $\sigma''(\vec{V_2}\vec{x}_{Q_0}) = \vec{x}_Q$ and σ and σ'' coincide on $fn(|Q_0|)\setminus\vec{V_2}\vec{x}_{Q_0}$. Let $\vec{V} = \vec{V_1}\cup\vec{V_2}$, then $E = (fn(|P_0|)\setminus\vec{V_1}\vec{x}_{P_0}) \cap (fn(|Q_0|)\setminus \vec{V_2}\vec{x}_{Q_0})$, and σ, σ' and σ'' coincide on E.

$\underline{\Leftarrow}$: write $\vec{V} = [V_1, \ldots, V_n]$, pick n new names $\vec{v} = [v_1, \ldots, v_n]$, and define $\sigma(V_i) = v_i$. Decompose $fn(|P_0|) \cup fn(|Q_0|)$ into the disjoint union:

$$\begin{array}{r} \vec{x}_{P_0} \ \cup\ \vec{x}_{Q_0} \ \cup\ \vec{V} \ \cup\ (fn(|P_0|) \setminus (fn(|Q_0|) \cup \vec{V}\vec{x}_{P_0})) \\ \cup\ (fn(|Q_0|) \setminus (fn(|P_0|) \cup \vec{V}\vec{x}_{Q_0}))\ \cup\ E\,, \end{array}$$

and define

$\sigma = \sigma'$ $(= \sigma'')$ on E;	$\sigma = \sigma'$ on $fn(\|P_0\|) \setminus (fn(\|Q_0\|) \cup \vec{x}_{P_0}\vec{V})$;
$\sigma(\vec{x}_{P_0}) = \vec{x}_{P_0}$ and $\sigma(\vec{x}_{Q_0}) = \vec{x}_{Q_0}$;	$\sigma = \sigma''$ on $fn(\|Q_0\|) \setminus (fn(\|P_0\|) \cup \vec{x}_{Q_0}\vec{V})$;
$\sigma(V_i) = v_i$ as seen above.	

As in the previous proof, use Lemma 3.2 to conclude. □

The characterisation given in Proposition 3.6 can be seen as an enlargement of the characterisation of the up-to injective substitutions case: the up-to restrictions technique compels σ' and σ'' to coincide on a smaller set E of names, or in other words more names in $fn(P_0)$ and $fn(Q_0)$ can be mapped to different names by σ' and σ''.

Here again, the method given by the characterisation is the following: to check if $(P, Q) \in \equiv (\mathcal{R}^i)^\nu \equiv$, compute $\mathsf{NR}(P)$ and $\mathsf{NR}(Q)$, and look for $(P_0, Q_0) \in \mathcal{R}$ such that the conditions of the characterisation hold.

About the lexicographical order on processes: The two checking methods above (as well as the characterisation of the next proof technique we study) rely on the ability, given $|\mathsf{NR}(P)|$ and $|P_0|$, to infer a substitution σ', injective on $fn(|P_0|)$, and such that $|\mathsf{NR}(P)|=_{\alpha\nu}|P_0|\sigma'$ (and similarly for σ''). This question is related to the definition of a lexicographical order on processes: consider for example $|\mathsf{NR}(P)| = !x \mid !y \mid \overline{x}$ and $|P_0| = !y \mid !x \mid \overline{x}$. Then, to infer a substitution matching $|\mathsf{NR}(P)|$ and $|P_0|$, we should look for a canonical way to present $!x \mid !y$, which has to be invariant under injective substitutions on names, and thus cannot depend on an order on names.

In general, we have not been able to define such a canonical ordering on processes, which means that we have to introduce some combinatorics and sometimes generate several substitutions, according on how normal processes having the same structure are ordered (by "processes having the same structure", we mean processes that are equal up to some (non necessarily

injective) renaming). We shall not enter here the details of our treatment of this problem; let us just say that the more powerful our technique is, the more permutations of "structurally equal processes" we have to take into account to apply our methods (i.e. the less narrow is the equivalence class associated to our order relation).

3.3 Up to Parallel Composition

To reason with both the up to restriction and the up to parallel composition proof techniques, we introduce the notion of *context*: a context (ranged over by C, C') is a term that contains a *hole*, written $[\,]$. If we want to combine the up to restriction and up to parallel composition proof techniques, we have to work with contexts that are described by the following syntax:

$$C \;=\; [\,] \mid (\nu x)\, C \mid C|P\,,$$

where P ranges over processes and x ranges over names. Given a context C and a process P, we construct the process $C[P]$ by replacing the hole with P.

Definition 3.7 *The closure under contexts of a relation $\mathcal{R}$ is written $\mathcal{R}^{\mathcal{C}}$ and defined by*

$$\mathcal{R}^{\mathcal{C}} \stackrel{def}{=} \{(P,Q); \exists C, P_0, Q_0 .\, P = C[P_0], Q = C[Q_0], P_0 \mathcal{R} Q_0\}\,.$$

As usual, we will work up to structural congruence, and consider $\equiv(\mathcal{R}^{\mathcal{C}})\equiv$, which is defined by replacing "$=$" with "$\equiv$" in the definition of $\mathcal{R}^{\mathcal{C}}$ (this actually justifies the fact that we authorise parallel composition of processes only at the right in the definition of contexts).

To work up to structural congruence will actually allow us to adopt a simpler form for contexts, ranged over by $\underline{C}$, and defined by the following syntax (we define $\mathcal{R}^{\underline{\mathcal{C}}}$ analogously as above.):

$$\underline{C} \;=\; (\nu \vec{x})\,([\,]|T) \qquad T = \prod_i (\alpha_i.N_i)^{m_i},\ T \text{ in normal form}\,.$$

Indeed, we have the following property:

Lemma 3.8 *For any relation $\mathcal{R}$, we have $\equiv(\mathcal{R}^{\mathcal{C}})\equiv \;=\; \equiv(\mathcal{R}^{\underline{\mathcal{C}}})\equiv$.*

Proof. The inclusion $\supseteq$ is obvious. For the other direction, consider $(P,Q) \in \equiv(\mathcal{R}^{\mathcal{C}})\equiv$, and show $(P,Q) \in \equiv(\mathcal{R}^{\underline{\mathcal{C}}})\equiv$. This means proving that for any context C and processes P, Q, P_0, Q_0 such that $P_0 \mathcal{R} Q_0$, $P \equiv C[P_0]$ and $Q \equiv C[Q_0]$, there exist a context $\underline{C}$ and processes P_1, Q_1 such that $P_1 \mathcal{R} Q_1$, $P \equiv \underline{C}[P_1]$ and $Q \equiv \underline{C}[Q_1]$. We proceed by induction over the structure of C; if $C = []$ or $C = (\nu x)\, C'$, the property is easy.

Consider the case where $C = C'|T_0$, with $\mathsf{NR}(T_0) = (\nu \vec{x_0})\,|U|$, for some $|U|$; by induction hypothesis, we know that there exist $\underline{C'}$, P_2 and Q_2 such that $P_2 \mathcal{R} Q_2$, $C'[P_2] \equiv \underline{C'}[P_2]$ and $C'[Q_2] \equiv \underline{C'}[Q_2]$. Write $C' = (\nu \vec{x})\,([] \mid |V|)$, for some $|V|$, then we have that

$$\begin{array}{rcll} C[P_2] & = & (\nu \vec{x})\,(P_2 \mid |V|) \mid (\nu \vec{x_0})\,|U| & \\ & \equiv & (\nu \vec{x}\vec{x_0})\,(P_2 \mid |V| \mid |U|) & \\ & \equiv & (\nu \vec{x}\vec{x_0})\,(P_2 \mid (|V| \oplus |U|)) & \text{(Lemma 1.7)}, \end{array}$$

which shows, reasoning along the same lines with Q_2, that $\underline{C}$, defined by $\underline{C} = (\nu \vec{x}\vec{x_0})\,([] \mid (|V| \oplus |U|))$ satisfies the required property (with processes P_2 and Q_2 satisfying $P_2 \mathcal{R} Q_2$), and this concludes the proof. □

Proposition 3.9 (Characterisation of $\equiv(\mathcal{R}^i)^{\underline{\mathcal{C}}}\equiv$) *Given a relation $\mathcal{R}$ and two processes P and Q, $(P,Q) \in \equiv(\mathcal{R}^i)^{\underline{\mathcal{C}}}\equiv$ iff there exist $(P_0, Q_0) \in \mathcal{R}$, a process $|T|$, two substitutions σ' and σ'' injective on $fn(|P_0| \mid |T|)$ and $fn(|Q_0| \mid |T|)$ respectively, and a name list $\vec{V} \subseteq fn(P_0) \cup fn(Q_0) \cup fn(|T|)$ such that, if we write $\mathsf{NR}(P) = (\nu \vec{x}_P)\,|\mathsf{NR}(P)|$, $P_0 = (\nu \vec{x}_{P_0})\,|P_0|$, $\vec{V}_1 = \vec{V}_{|fn(P_0)}$, $|T_1|_\omega =_{\alpha\nu} |T|_\omega \setminus |P_0|_\omega$, $|T_1|_{\mathcal{N}} =_{\alpha\nu} |T|_{\mathcal{N}} \setminus |P_0|_\omega$, and similarly for Q, Q_0, $\vec{x}_{Q_0}$, $\vec{V}_2$ and $|T_2|$, we have:*

$$(i)\ \begin{cases} |\mathsf{NR}(P)|_\omega =_{\alpha\nu} (|P_0|_\omega \mid |T_1|_\omega)\sigma' \\ |\mathsf{NR}(Q)|_\omega =_{\alpha\nu} (|Q_0|_\omega \mid |T_2|_\omega)\sigma'' \end{cases}$$

$$\text{and}\ \ (i')\ \begin{cases} |\mathsf{NR}(P)|_{\mathcal{N}} =_{\alpha\nu} (|P_0|_{\mathcal{N}} \setminus |T_1|_\omega \mid |T_1|_{\mathcal{N}})\sigma' \\ |\mathsf{NR}(Q)|_{\mathcal{N}} =_{\alpha\nu} (|Q_0|_{\mathcal{N}} \setminus |T_2|_\omega \mid |T_2|_{\mathcal{N}})\sigma'' \end{cases}$$

$$(ii)\ \sigma'(\vec{V}_1 \vec{x}_{P_0}) = \vec{x}_P \ \ \text{and} \ \ \sigma''(\vec{V}_2 \vec{x}_{Q_0}) = \vec{x}_Q$$

$$(iii)\ \sigma' = \sigma''\ \text{on}\ E = fn(|P_0| \oplus |T|) \cap fn(|Q_0| \oplus |T|) \setminus (\vec{V}\, \vec{x}_{P_0} \vec{x}_{Q_0})\,.$$

The proof of this result follows the lines of the proofs of Propositions 3.4 and 3.6. With respect to the latter proof, the novelty comes from the fact that we have to do some reasoning about the bodies of the processes as we

normalise them, instead of taking them as they are; this is easily done using Lemma 1.7. We will not enter here the details of this proof, but instead give a description of the checking method induced by the result above.

Given two processes P and Q, and a relation $\mathcal{R}$, to check if $(P,Q) \in \equiv (\mathcal{R}^i)\underline{\mathcal{C}}\equiv$:

1. compute $\mathtt{NR}(P)$ and $\mathtt{NR}(Q)$, yielding $\mathtt{NR}(P) = (\nu\vec{x}_P)\,|\mathtt{NR}(P)|$ and $\mathtt{NR}(Q) = (\nu\vec{x}_Q)\,|\mathtt{NR}(Q)|$;
2. pick $(P_0, Q_0) \in \mathcal{R}$, and write $P_0 = (\nu\vec{x}_{P_0})\,|P_0|$ and $Q_0 = (\nu\vec{x}_{Q_0})\,|Q_0|$;
3. use (i) to compute $|T_1|_\omega$ and σ', and $|T_2|_\omega$ and σ'', and check that we can define $|T|_\omega$ s.t. $|T_1|_\omega = |T|_\omega \setminus |P_0|_\omega$ and $|T_2|_\omega = |T|_\omega \setminus |Q_0|_\omega$;
4. proceed similarly with (i') to compute $|T_1|_\mathcal{N}$, $|T_2|_\mathcal{N}$ and $|T|_\mathcal{N}$, possibly extending the domains of σ' and σ'';
5. check conditions (ii) and (iii) on σ' and σ''.

As hinted above, the generation of σ' and σ'' can involve some combinatorics that heavily rely on the definition of a lexicographical order on processes.

4 An Implementation

4.1 The Tool

We present a prototype implementation of the methods described in the latter Section, under the form of a tool for checking bisimulation using the up to techniques. This tool, written in O'Caml, allows the user to define a pair of processes, choose an up-to technique among those studied above, and try to prove bisimilarity using this technique. In the case where the proof succeeds, the corresponding bisimulation relation is displayed; if the processes are not bisimilar, some kind of diagnostic information is given to the user to justify the failure (and hopefully help him make another attempt). Other features, like the computation of the normal form of a process and the interactive simulation of the behaviour of a process, are also provided.

Note that the tool allows the user to check also *weak bisimilarity* (written $\approx$), that is defined by replacing $Q \xrightarrow{\mu} Q'$ with $Q \stackrel{*}{\Rightarrow} Q'$ if $\mu = \tau$, with $Q \stackrel{*}{\Rightarrow}\xrightarrow{\mu}\stackrel{*}{\Rightarrow} Q'$ if $\mu \neq \tau$, in Definition 2.1, where $\stackrel{*}{\Rightarrow}$ denotes the reflexive, transitive closure of $\xrightarrow{\tau}$. All the up-to techniques we have studied, as well as our results, extend directly to the weak case.

Algorithm To each proof technique $\mathcal{F}$ we have seen in Section 3 corresponds a decision procedure $\mathtt{decide}_\mathcal{F}$, given by the characterisations of Propositions 3.4, 3.6 and 3.9. Our "bisimulation up-to $\mathcal{F}$" checking function $\mathtt{bisim}_\mathcal{F}$ takes three arguments, namely a relation $\mathcal{R}$ and two processes P

and Q, and returns an up-to $\mathcal{F}$ bisimulation relation extending $\mathcal{R}$ that contains P and Q. It basically sticks to the definition of an up-to bisimulation given in Definition 2.2, trying to build up an up-to bisimulation until it reaches a fixpoint.

To compute $\texttt{bisim}_{\mathcal{F}}(\mathcal{R}, P, Q)$*:*

- *(parameter:* $\mathcal{R}$*) pick a transition* $P \xrightarrow{\mu} P'$ *of* P*, and compute* $Q_\mu = \{Q'. Q \xrightarrow{\mu} Q'\}$*;*

  - use $\texttt{decide}_{\mathcal{F}}$ *to check if any of the elements of* Q_μ *satisfies* $P'\mathcal{F}(\mathcal{R})Q'$*. If such an element can be found, loop to another transition of* P*, leaving* $\mathcal{R}$ *unchanged;*

  - otherwise, pick a $Q' \in Q_\mu$ *and make the recursive call* $\texttt{bisim}_{\mathcal{F}}((P', Q') :: \mathcal{R}, P', Q')$*; if this call succeeds, yielding* $\mathcal{R}$*', loop to another transition of* P *with* $\mathcal{R}$*', otherwise pick another* $Q' \in Q_\mu$*; if all the recursive calls to* $\texttt{bisim}_{\mathcal{F}}$ *fail, fail;*
- *proceed similarly with the transitions of* Q*.*

Figure 4: The checking algorithm

Our checking algorithm is informally described on Figure 4. Its correctness follows from the soundness of the closure functions we apply to relations, as proved in [San95]. Of course, our algorithm is not complete, since in the case where the candidate bisimulation relation we generate keeps growing even up to the techniques we use, the program enters an infinite loop.

4.2 Examples

We give here a few simple examples to illustrate the way our tool works.

- $(\nu b)(!b.a(x).\overline{x} \mid !a(t).\overline{t} \mid !\overline{b}) \sim !a(x).\overline{x} \mid (\nu c)\,(!\overline{c} \mid !c)$: in the proof of this result, the normalisation algorithm erases each copy of $a(x).\overline{x}$ that is generated after a communication over b takes place in the left hand side process. We show a simple session, where the user defines the left and right processes (commands `L` and `R`), asks the system to print the pair of processes (command `P`), and checks bisimilarity (command `C`):

```
> L (^b) (!b.a(x).x[] | !a(t).t[] | !b[] )
> R !a(x).x[] | (^c)(!c[] | !c)
> P
The pair is
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]) , (!a(x).x[] | (^c)(!c[] | !c)))
> C
Yes, size of the relation is 1
```

```
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]),(^c)(!c | !a(x).x[] | !c[]));
```

Both these processes are weakly bisimilar to $!a(x).\overline{x}$; here the user uses command `S` to toggle the bisimilarity checking mode (from strong to weak):

```
> P
The pair is
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]) , !a(x).x[])
> S
Checking mode is weak, verbose mode is on.
> C
Yes, size of the relation is 1
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]),!a(x).x[]);
```

- $(\nu a)\,(!\overline{a}b \mid !a(x).\overline{x}) \sim (\nu c)\,(!\overline{c} \mid !c.\overline{b})$: here, when both processes perform a τ action, a copy of $\overline{b}$ is liberated on each side, and can be immediately erased; without the up to parallel composition proof technique, the relation would keep growing, generating copies of $\overline{b}$.

```
> C
Yes, size of the relation is 1
((^a)(!a(x).x[] | !a[b]),(^c)(!c.b[] | !c[]));
```

- Another law, which is a straightforward instantiation of the so-called *replication theorems*, that express the distributivity of private resources: $(\nu a)\,(!a(x).\overline{x} \mid !\overline{a}b \mid !\overline{a}c) \;\sim\; (\nu a)\,(!a(x).\overline{x} \mid !\overline{a}b) \mid (\nu a)\,(!a(x).\overline{x} \mid !\overline{a}b)$ (processes $!\overline{a}b$ and $!\overline{a}c$ can either share a common resource $!a(x).\overline{x}$ - that sends a signal on the name it receives on a -, or have their own copy of this resource; note the shape of the normal form for the right hand side process):

```
> C
Yes, size of the relation is 1
((^a)(!a(x).x[] | !a[c] | !a[b]),
(^e')(^a)(!a(x).x[] | !e'(x).x[] | !a[c] | !e'[b]));
```

Conclusion

We have developed some methods to automatically check bisimilarities between π-calculus processes, and shown their expressive power on a prototype implementation[4].

[4] A beta version of the tool is available at `http://cermics.enpc.fr/~dh/pi/`

Related work The Mobility Workbench [VM94] is the system that is probably the closest to ours, and has already been discussed above. Other tools for checking bisimulations over process algebra terms include Cesar/Aldebaran [JCFS96], the Jack Toolkit [S.G94], and the FC2tools package [Sim96]. These systems are robust and include techniques to increase efficiency (such as Binary Decision Diagrams), as well as visual interfaces for the representation of complex systems. The methods they use (partition refinement algorithm, on the fly proof method, compositional reductions and abstraction) are based on a semantical representation that is computed from the process terms (like Labelled Transition Systems or automata), and can thus handle only finite states processes; on the contrary, our tool remains at the level of syntax, and should be seen as a first step towards a promising direction regarding the study of infinite states processes, rather than as a challenger for these systems in terms of efficiency.

The notion of normal form appears in the literature through *axiomatisations*; axiomatisations of finite control processes have been given for example for *open* bisimulation [San96b] (this axiomatisation is used in the Mobility WorkBench), as well as for the fusion calculus [PV98], that is a promising language for the task of the implementation of verification methods. For replicated terms in the general case, [EG96] proves decidability for an extended version of structural congruence; in this work, any form of process can be replicated; we have chosen a smaller language to keep the reasoning about the up-to techniques more clear.

Future work A key theoretical issue that has to be studied regarding our techniques is *completeness*. Our procedure is not complete: even for terms that can be proved bisimilar using a finite up-to relation, we may in some cases enter an infinite loop, because of "'blind" recursive calls to the bisimulation checking function. One could avoid this by modifying the algorithm of Figure 4 to adopt a breadth-first strategy, thus reaching some form of computational completeness. More significantly, one is interested in finding a non-trivial class of processes (containing infinite-state terms) for which the up-to bisimulation proof framework gives a decision procedure.

Our system can be enhanced in many ways:

The syntax of processes could be enriched by adding the choice construct $(+)$ and recursive definitions to our language; this would require the adaption of our methods to the extended language. We may alternatively choose to use well-known encodings [Nes97, Mil91], which would compel the user to work only with weak bisimilarity.

The size of the relations could be reduced by enriching structural congruence with additional laws (e.g. $(x \notin n(\alpha)) \Rightarrow (\nu x)\,\alpha.P \equiv \alpha.(\nu x)\,P$), and

adapting our normalisation algorithm, as well as by considering other up-to techniques.

Some amount of interaction in the bisimilarity proofs could also be introduced, by allowing the definition of breakpoints in the construction of the bisimulation relation, in order to "help" the tool avoiding infinite loops (for example by applying bisimilarity laws that do not belong to structural congruence).

Another interesting direction could be to adapt our methods to *open terms*, in order to be able to prove not only bisimilarity results, but also general bisimilarity laws. To this aim, relevant works include [Ren97] and [Sim85].

Finally, the proofs of this paper could be mechanised reusing the work of [Hir97], which could allow one, using *reflection* [Bou97], to extract a *certified* bisimilarity checker. It seems sensible to think, however, that some more work has to be done in order to make these proofs tractable for the purpose of a theorem prover formalisation.

Acknowledgements Many thanks go to Michele Boreale for constant help during this study, as well as to Davide Sangiorgi for introducing the theoretical basis of it, and for insightful discussions.

References

[Bou97] S. Boutin. Using reflection to build efficient and certified decision procedures. In Martin Abadi and Takahashi Ito, editors, *TACS'97*, volume 1281. LNCS, Springer-Verlag, 1997.

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proceedings of LICS '98 - to appear*, 1998.

[EG96] J. Engelfriet and T. Gelsema. Multisets and structural congruence of the π-calculus with replication. Report 2/95, Leiden University, 1996.

[Hir97] D. Hirschkoff. A full formalisation of π-calculus theory in the Calculus of Constructions. In *Proceedings of TPHOL'97*, volume 1275, pages 153–169. LNCS, Springer Verlag, 1997.

[JCFS96] A. Kerbrat R. Mateescu L. Mounier J.-C. Fernandez, H. Garavel and M. Sighireanu. Cadp (caesar/aldebaran development package): A protocol validation and verification toolbox. In *Proceed-*

ings of CAV' 96, volume 1102 of *LNCS - Springer Verlag*, pages 437–440, 1996.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, October 1991. Also in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer and H. Schwichtenberg, Springer-Verlag, 1993.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[Nes97] U. Nestmann. What is a 'good' encoding of guarded choice? In *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*, 19997.

[PS96] M. Pistore and D. Sangiorgi. A partition refinement algorithm for the π-calculus. In Rajeev Alur, editor, *Proceedings of CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, 1996.

[PT87] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.

[PV98] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proceedings of LICS' 98*, 1998. to appear.

[Ren97] A. Rensink. Bisimilarity of open terms. In C. Palamidessi and J. Parrow, editors, *Expressiveness in Concurrency*, 1997. also available as technical report 5/97, University of Hildesheim, may 1997.

[San95] D. Sangiorgi. On the bisimulation proof method. Revised version of Technical Report ECS–LFCS–94–299, University of Edinburgh, 1994. An extended abstract can be found in Proc. of MFCS'95, LNCS 969, 1995.

[San96a] D. Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report RR-3000, INRIA, 1996. to appear in Information and Computation.

[San96b] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996. Earlier version published as Report ECS-LFCS-93-270, University of Edinburgh. An extended abstract appeared in the *Proceedings of CONCUR '93*, LNCS 715.

[S.G94] A.Bouali S.Larosa S.Gnesi. The integration project in the JACK Environement. *EATCS Bulletin*, (54), 1994.

[Sim85] R. De Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Computer Science*, (37):245–267, 1985.

[Sim96] A.Bouali A.Ressouche V.Roy R.de Simone. The fc2 toolset. demo presentation at TACAS'96, AMAST'96 and CAV'96, 1996.

[SM92] D. Sangiorgi and R. Milner. Techniques of "weak bisimulation up to". In *CONCUR '92*, number 630 in LNCS, 1992.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *Proceedings of CAV'94*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.