Electronic Notes in Theoretical Computer Science 18 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume18.html 15 pages

Automatically Proving Up–to Bisimulation

Daniel Hirschkoff

CERMICS – ENPC/INRIA, F–77455 Marne la Vallée Cedex 2, France

Abstract

We present a tool 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 usefulness of our results on a prototype implementation.

Introduction

We present a tool for automatically checking bisimilarities between π–calculus processes with the up–to techniques for bisimulation. 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 ([9], used for example in [8]) 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* [14] 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 [13,11], and to our knowledge have only been used in papers about the theory of π–calculus, to prove bisimilarity

laws. They allow the user to perform some syntactical manipulations on processes in order to reduce the size of the relations one has to exhibit to prove bisimulation. The aim of this work is to investigate to what extent these techniques can be mechanised, and how such theoretical tools can be used in the context of verification. The main benefit we get from these techniques arises at the level of expressiveness: we can indeed prove in some cases bisimilarity between replicated processes having an infinite states space.

The basis for the development of our up to checking methods is the up to structural congruence proof technique. To work up to structural congruence means to have a notion of unique normal form for this equivalence. We achieve this in a simple way through the definition of a term rewriting system on a small but expressive language. This allows us to work only with terms in normal form, and to introduce effective characterisations of the various proof techniques we study (up to injective substitutions on names, up to restriction, and up to parallel composition). We rely on these characterisations to define our bisimilarity checking algorithm, which is a straightforward extension of the "on the fly" checking method for bisimulation.

For the sake of brevity, the presentation of the definitions and of the main results has been kept rather succinct (the reader should refer to [6] for the proofs of the properties we state and for more comments on the design choices). Insight on the technical issues that arise as we mechanise the up to techniques is given along the statements of our results. The paper is organised as follows: Section 1 describes the general framework of our study: after defining syntax and structural congruence, we introduce a normalisation algorithm that enjoys the uniqueness of normal form property. We then introduce semantics and the behavioural equivalence we use on processes, namely bisimilarity, and give a brief account on the up to proof techniques for bisimulation. In Section 2, we define the up to techniques we use, and give characterisations of the corresponding functions on relations. We put together these results in Section 3 to build a prototype implementation, and illustrate the behaviour of our techniques on a few simple examples. We finally conclude with a brief discussion on the insight brought by our study, and on future work.

1 Definitions and Notations

1.1 Syntax

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

$$\alpha \;=\; a(\boldsymbol{b}) \mid \overline{a}[\boldsymbol{b}]\,, \qquad\qquad P \;=\; \mathbf{0} \mid \alpha.P \mid !\alpha.P \mid (\nu x)\,P \mid P_1|P_2\,.$$

Prefixes are either input: $a(\boldsymbol{b})$, or output: $\overline{a}[\boldsymbol{b}]$. $\mathbf{0}$ is the inactive process;

prefixed processes are either linear ($\alpha.P$) or replicated ($!\alpha.P$); the other constructors are restriction (ν) and parallel composition ($|$). Bound names are defined by saying that restriction and input prefix are binding operators: (νx) and $x(\boldsymbol{y})$ respectively bind name x and the names in $\boldsymbol{y}$ in their continuation. 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 2, 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 $\dfrac{x \notin \mathrm{fn}(P)}{(\nu x)\,P \equiv P}$	6 $\dfrac{x \notin \mathrm{fn}(P)}{P\|(\nu x)\,Q \equiv (\nu x)\,(P\|Q)}$
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 [7], 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 $(\nu\boldsymbol{x})\,P$ for restriction, where $\boldsymbol{x}$ is intended as having a set rather than a vector structure (in order to allow silent applications of rule 4). The technical issues raised by the implementation of these aspects of structural congruence will be discussed in Section 2.

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. Remark that structural congruence preserves free names, i.e if $P \equiv Q$, then $\mathrm{fn}(P) = \mathrm{fn}(Q)$.

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.

1.2 Normal Forms

We give an orientation to the relevant structural congruence laws to define a term rewriting system as follows:

Definition 1.1 (Normalisation algorithm) *The normalisation algorithm is defined as the rewriting system given by the five following rules* [1] *:*

$$R5\ (\nu x)\,\mathbf{0} \to \mathbf{0} \qquad R6\ (x \notin \mathrm{fn}(P)) \Rightarrow (\,P|(\nu x)\,Q \to\ (\nu x)\,(P|Q)\,)$$

$$R1\ P|\mathbf{0} \to\ P \qquad R7\ !\alpha.P|\alpha.P \to !\alpha.P \qquad R8\ !\alpha.P|!\alpha.P \to !\alpha.P$$

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:

Proposition 1.2 (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 $\mathsf{NR}(P) =_{\alpha\nu} \mathsf{NR}(Q)$.*

We now give a syntactical description of the terms that cannot be rewritten by our algorithm:

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

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

Fig. 1. Syntax of normal forms

Proposition 1.3 (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.*

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

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

two components, its body and its topmost restrictions. The body is made of processes that are ready to commit, and the topmost restrictions define some kind of geometry among these processes.

Notations: Given a non–null normal process $P = (\nu \boldsymbol{x}_P) \prod_i (\alpha_i.N_i)^{m_i}$, we write $P = (\nu \boldsymbol{x}_P)\langle P \rangle$ to decompose P into its uppermost restrictions $(\nu \boldsymbol{x}_P)$ 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 $\langle P \rangle, \langle Q \rangle, \ldots$. We further decompose $\langle P \rangle$ into an "infinite part", written $\langle P \rangle_\omega$, and a "finite part", written $\langle P \rangle_{\mathcal{N}}$, respectively corresponding to the replicated and the non–replicated components, i.e. $\langle P \rangle_\omega = \prod_{i.\, m_i = \omega} (\alpha_i.N_i)^{m_i}$ and $\langle P \rangle_{\mathcal{N}} = \prod_{i.\, m_i \in \mathcal{N}} (\alpha_i.N_i)^{m_i}$.

We introduce some machinery on processes of the form $\langle P \rangle$, that will be useful for the treatment of the up to parallel composition proof technique in Section 2: 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 $\langle P \rangle_\omega$), and, for two processes of the form $\langle P \rangle$ and $\langle Q \rangle$, we let:

$$\langle P \rangle \oplus \langle Q \rangle \stackrel{def}{=} \langle P \rangle_\omega \mid (\langle Q \rangle_\omega \setminus \langle P \rangle_\omega) \mid (\langle P \rangle_{\mathcal{N}} \setminus \langle Q \rangle_\omega) \mid (\langle Q \rangle_{\mathcal{N}} \setminus \langle P \rangle_\omega)\,.$$

1.3 Semantics

1.3.1 Operational Semantics and Bisimulation

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

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

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

$$\text{CLOSE}_1\ \frac{P \xrightarrow{a(\boldsymbol{b})} P' \qquad Q \xrightarrow{(\nu \boldsymbol{b'})\,\overline{a}[\boldsymbol{b}]} Q'}{P|Q \xrightarrow{\tau} (\nu \boldsymbol{b'})\,(P'|Q')}\ \ \boldsymbol{b'} \cap \mathrm{fn}(P) = \emptyset$$

Fig. 2. Early Transition Semantics

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

Definition 1.4 (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.*

1.3.2 The up–to Proof Techniques for Bisimulation

To rephrase Definition 1.4 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.

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

Fig. 3. From bisimulation to up–to bisimulation

In [11], 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 technique is represented by a functional from relations to relations (ranged over by $\mathcal{F}$):

Definition 1.5 (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 $\mathcal{F}$ gives a correct proof technique if it is *sound*, i.e. if ($\mathcal{R}$ is a bisimulation up to $\mathcal{F}$) implies ($\mathcal{R} \subseteq \sim$), which means that 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). [11] 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 can be combined together for the task of proving bisimulations, thanks to nice compositionality properties of respectful functions.

2 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. Using Definition 1.5, this amounts to define a checking method to decide if a relation is a bisimulation up to $\mathcal{F}$.

The normalising function we have defined is naturally used in the framework of the bisimulation up to bisimilarity proof technique. We now study the up to injective substitutions on free names, the up to restriction and the up to parallel composition proof techniques. The corresponding functions are proved correct in [11]; however, their nice compositionality properties do not extend to our characterisations, which prevents us to treat them separately. Therefore, we shall treat them incrementally, adding a technique at each step, which will lead to our most powerful technique in Definition 2.7. Notice anyway that the overall methodology of our checking methods is rather uniform. Because of lack of space, we will just state the definitions and characterisations of the various proof techniques (the reader interested in a more detailed discussion and in the proofs should refer to [6]), and then comment about the algorithms induced by our characterisations.

2.1 Definitions and Characterisations

We first need some background on substitutions on names, that are functions from names to names, ranged over by σ, σ', σ''. We define $\mathrm{dom}(\sigma)$, the domain of σ, as the set of names n such that $\sigma(n) \neq n$, and the codomain $\mathrm{codom}(\sigma)$ of σ as $\sigma(\mathrm{dom}(\sigma))$. 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. $\mathrm{codom}(\sigma) \cap \mathrm{bn}(P) = \emptyset$ (this is always possible using α–conversion). An injective substitution σ whose domain is finite defines a bijective mapping between $\mathrm{dom}(\sigma)$ and $\mathrm{codom}(\sigma)$; we shall write in this case σ^{-1} for the inverse of σ. Given a set of names E, we say that two substitutions σ and σ' *coincide* on E, written $\sigma = \sigma'$ on E, iff for any name n in E, $\sigma(n) = \sigma'(n)$.

2.1.1 Up to Injective Substitutions on Names

Definition 2.1 *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:*

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

In order to give a characterisation of $\equiv\mathcal{R}^i\equiv$, we study α–convertibility of processes in normal form, and its relation to injective substitutions:

Remark 2.2 *Let P and Q be two processes in normal form; write $P = (\nu \boldsymbol{x})\langle P\rangle$ and $Q = (\nu \boldsymbol{y})\langle Q\rangle$. Then $(\nu \boldsymbol{x})\langle P\rangle =_{\alpha\nu} (\nu \boldsymbol{y})\langle Q\rangle$ iff there exists a substitution σ, injective on $\mathrm{fn}(\langle Q\rangle)$, s.t. $\langle P\rangle =_{\alpha\nu} \langle Q\rangle\sigma$, $\mathrm{dom}(\sigma) = \boldsymbol{y}$ and $\sigma(\boldsymbol{y}) = \boldsymbol{x}$.*

Lemma 2.3 *Write as above $P = (\nu \boldsymbol{x})\langle P\rangle$ and $Q = (\nu \boldsymbol{y})\langle Q\rangle$, for two normal processes P and Q. Then, for any substitution σ injective on the free names of $\langle Q\rangle$, $(\nu \boldsymbol{x})\langle P\rangle =_{\alpha\nu} (\nu \boldsymbol{y})(\langle Q\rangle\sigma)$ iff there exists a substitution σ' injective on $\mathrm{fn}(\langle Q\rangle)$ s.t. (i) $\langle P\rangle =_{\alpha\nu} \langle Q\rangle\sigma'$, (ii) $\sigma'(\sigma^{-1}(\boldsymbol{y})) = \boldsymbol{x}$, and (iii) $\sigma' = \sigma$ on $\mathrm{fn}(\langle Q\rangle) \setminus \sigma^{-1}(\boldsymbol{y})$.*

Proposition 2.4 (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 $\mathrm{fn}(\langle P_0\rangle)$ and $\mathrm{fn}(\langle Q_0\rangle)$ respectively, such that, if we write:

$P_0 = (\nu \boldsymbol{x}_{P_0})\langle P_0\rangle$, $Q_0 = (\nu \boldsymbol{x}_{Q_0})\langle Q_0\rangle$, $\mathsf{NR}(P) = (\nu \boldsymbol{x}_P)\langle \mathsf{NR}(P)\rangle$ and $\mathsf{NR}(Q) = (\nu \boldsymbol{x}_Q)\langle \mathsf{NR}(Q)\rangle$, we have:

$P_0 \mathcal{R} Q_0$, $\langle \mathsf{NR}(P)\rangle =_{\alpha\nu} \langle P_0\rangle\sigma'$, $\langle \mathsf{NR}(Q)\rangle =_{\alpha\nu} \langle Q_0\rangle\sigma''$, $\sigma'(\boldsymbol{x}_{P_0}) = \boldsymbol{x}_P$, $\sigma''(\boldsymbol{x}_{Q_0}) = \boldsymbol{x}_Q$, and, if we write $E = \mathrm{fn}(\langle P_0\rangle) \cap \mathrm{fn}(\langle Q_0\rangle) \setminus \boldsymbol{x}_{P_0}\boldsymbol{x}_{Q_0}$, $\sigma' = \sigma''$ on E.

2.1.2 Up to Restrictions

We now turn to the up to restrictions proof technique.

Definition 2.5 *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:*

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

Proposition 2.6 (Characterisation of $\equiv(\mathcal{R}^i)^\nu\equiv$)
Given two processes P and Q and a relation $\mathcal{R}$, write $\mathsf{NR}(P) = (\nu \boldsymbol{x}_P)\langle \mathsf{NR}(P)\rangle$ and $\mathsf{NR}(Q) = (\nu \boldsymbol{x}_Q)\langle \mathsf{NR}(Q)\rangle$. Then $(P, Q) \in \equiv(\mathcal{R}^i)^\nu\equiv$ iff there exist two substitutions σ' and σ'' injective on $\mathrm{fn}(\langle P_0\rangle)$ and $\mathrm{fn}(\langle Q_0\rangle)$ respectively, processes P_0 and Q_0 and a name list $\boldsymbol{V} \subseteq \mathrm{fn}(P_0) \cup \mathrm{fn}(Q_0)$ such that, if we write $P_0 = (\nu \boldsymbol{x}_{P_0})\langle P_0\rangle$, $Q_0 = (\nu \boldsymbol{x}_{Q_0})\langle Q_0\rangle$, $\boldsymbol{V}_1 = \boldsymbol{V}_{|\mathrm{fn}(P_0))}$ and $\boldsymbol{V}_2 = \boldsymbol{V}_{|\mathrm{fn}(Q_0)}$:

$$(i)\ \langle \mathsf{NR}(P)\rangle =_{\alpha\nu} \langle P_0\rangle\sigma' \text{ and } \langle \mathsf{NR}(Q)\rangle =_{\alpha\nu} \langle Q_0\rangle\sigma''$$
$$(ii)\ \sigma'(\boldsymbol{V}_1 \boldsymbol{x}_{P_0}) = \boldsymbol{x}_P \text{ and } \sigma''(\boldsymbol{V}_2 \boldsymbol{x}_{Q_0}) = \boldsymbol{x}_Q$$
$$(iii)\ \sigma' = \sigma'' \text{ on } E = \mathrm{fn}(\langle P_0\rangle) \cap \mathrm{fn}(\langle Q_0\rangle) \setminus (\boldsymbol{V} \boldsymbol{x}_{P_0} \boldsymbol{x}_{Q_0}).$$

The condition above can be seen as an enlargement 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 $\mathrm{fn}(P_0)$ and $\mathrm{fn}(Q_0)$ can be mapped to different names by σ' and σ''.

2.1.3 Up to Parallel Composition

To reason with both the up to restriction and the up to parallel composition proof techniques, we work with contexts that are described by the following syntax (note that the up to structural congruence proof technique allows us to adopt this simple form of contexts without loss of generality):

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

Definition 2.7 *We write $\equiv(\mathcal{R}^i)^{\mathcal{C}}\equiv$ to denote the closure under injective substitutions, structural congruence, restriction and parallel composition of a relation $\mathcal{R}$, defined as follows:*

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

Proposition 2.8 (Characterisation of $\equiv(\mathcal{R}^i)^{\mathcal{C}}\equiv$) *Given a relation $\mathcal{R}$ and two processes P and Q, $(P,Q)\in\equiv(\mathcal{R}^i)^{\mathcal{C}}\equiv$ iff there exist $(P_0,Q_0)\in\mathcal{R}$, a process $\langle T\rangle$, two substitutions σ' and σ'' injective on $\mathrm{fn}(\langle P_0\rangle \mid \langle T\rangle)$ and $\mathrm{fn}(\langle Q_0\rangle \mid \langle T\rangle)$ respectively, and a name list $\boldsymbol{V}\subseteq \mathrm{fn}(P_0)\cup\mathrm{fn}(Q_0)\cup\mathrm{fn}(\langle T\rangle)$ such that, if we write $\mathbf{NR}(P)=(\nu\boldsymbol{x}_P)\langle\mathbf{NR}(P)\rangle$, $P_0=(\nu\boldsymbol{x}_{P_0})\langle P_0\rangle$, $\boldsymbol{V}_1=\boldsymbol{V}_{|\mathrm{fn}(P_0)}$, $\langle T_1\rangle_\omega{=}_{\alpha\nu}\langle T\rangle_\omega\setminus\langle P_0\rangle_\omega$, $\langle T_1\rangle_{\mathcal{N}}{=}_{\alpha\nu}\langle T\rangle_{\mathcal{N}}\setminus\langle P_0\rangle_\omega$, and similarly for Q, Q_0, $\boldsymbol{x}_{Q_0}$, $\boldsymbol{V}_2$ and $\langle T_2\rangle$, we have:*

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

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

$$(ii)\ \sigma'(\boldsymbol{V}_1\boldsymbol{x}_{P_0})=\boldsymbol{x}_P \text{ and } \sigma''(\boldsymbol{V}_2\boldsymbol{x}_{Q_0})=\boldsymbol{x}_Q,\\ \boldsymbol{x}_{\boldsymbol{P_0}}\cap\mathrm{fn}(\langle T_1\rangle)=\boldsymbol{x}_{\boldsymbol{Q_0}}\cap\mathrm{fn}(\langle T_2\rangle)=\emptyset,$$

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

2.2 On the induced checking methods

Let us now describe informally the implementation of the checking methods induced by Propositions 2.4, 2.6 and 2.8. The overall methodology is the same in each case; we illustrate it on the *second* proof technique. Given two processes P and Q, and a relation $\mathcal{R}$, to decide if $(P,Q)\in\equiv(\mathcal{R}^i)^\nu\equiv$:

1 compute the normal forms of P and Q, yielding $\mathbf{NR}(P)=(\nu\boldsymbol{x}_P)\langle\mathbf{NR}(P)\rangle$ and $\mathbf{NR}(Q)=(\nu\boldsymbol{x}_Q)\langle\mathbf{NR}(Q)\rangle$;

2 pick $(P_0,Q_0)\in\mathcal{R}$, and compute as above $P_0=(\nu\boldsymbol{x}_{P_0})\langle P_0\rangle$ and $Q_0=(\nu\boldsymbol{x}_{Q_0})\langle Q_0\rangle$; if any of the checks in steps 3 and 4 fails, go back to point

2 with another pair (P_0, Q_0) of processes;

3 use (i) to compute σ' s.t. $\langle \mathbf{NR}(P)\rangle =_{\alpha\nu} \langle P_0\rangle\sigma'$, and proceed similarly for σ'';

4 find $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ and extend σ' and σ'' so that (ii) and (iii) hold.

Steps 1 and 2 use the normalisation function of Section 1; the difficulties are concentrated in step 3, where we have to derive matchings between two bodies of processes, and infer the corresponding substitutions on free names. Indeed, in our mathematical reasoning, we treat process bodies as parallel compositions whose components can be implicitly rearranged the way we want (see above). When it comes to implementation, however, we have to front the question of the ordering of parallel components, which is not an easy task. Consider for example the case where $\langle \mathbf{NR}(P)\rangle = !a \mid !b \mid \overline{a}$ and $\langle P_0\rangle = !y \mid !x \mid \overline{x}$: here the two bodies can be matched together by associating a to x and b to y, but, in general, we also have to check that the matching $a \to y$, $b \to x$ fails (this matching comes from the comparison of $!a \mid !b$ and $!y \mid !x$, that are made of processes equal up to renaming). Indeed, while we can impose that the $\overline{x}$ component comes after the replicated input components, because we can distinguish these subterms "structurally", we have not been able to define a canonical ordering for a parallel composition such as $!x \mid !y$. Such an order on processes should in effect be invariant under injective substitution on free names (a proof technique that is involved in all our checking methods), which makes an ordering based on an order on names hopeless. Therefore, we cannot find an easy way to ordinate two processes that are equal up to some renaming, and we are compelled to introduce some combinatorics on the lists of parallel components of a process. In our example, we have to take into account the two possible orderings of $!x$ and $!y$ in order to be sure to infer the matching between $\langle \mathbf{NR}(P)\rangle$ and $\langle P_0\rangle$.

Note as well that, in addition to the problem of name matching, conditions (i) and (i') also require the definition of a property of *structural inclusion* (akin to [3]), meaning that a process is a subpart of a parallel composition (to isolate term T). This extra requirement involves a further use of combinatorics to explore all possible matchings and inclusions.

3 An Implementation

3.1 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 1.4, 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.

The algorithm To each proof technique $\mathcal{F}$ we have seen in Section 2 corresponds a decision procedure $\texttt{decide}_{\mathcal{F}}$, given by the characterisations of Propositions 2.4, 2.6 and 2.8. Our "bisimulation up–to $\mathcal{F}$" checking function $\texttt{bisim}_{\mathcal{F}}$ (defined on Figure 4) takes three arguments: 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, Q). The computation of this function follows Definition 1.5, by trying to build up an up–to bisimulation until it reaches a fixpoint. Its correctness derives from the soundness of the closure functions we apply to relations, as proved in [11]. Of course, our algorithm is not complete, since in the case where the candidate bisimulation relation keeps growing even up to the techniques we use, the program enters an infinite loop.

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.

Fig. 4. The checking algorithm

3.2 Examples

- $(\nu b)(!b.a(x).\overline{x} \mid !a(t).\overline{t} \mid !\overline{b}) \sim !a(x).\overline{x} \mid (\nu c)\,(!\overline{c} \mid !c)$: 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 `Left` and `Right`), asks the system to print the pair of processes (command `Print`), and checks bisimilarity (command `Check`):

```
> Left (^b) (!b.a(x).x[] | !a(t).t[] | !b[] )
> Right !a(x).x[] | (^c)(!c[] | !c)
> Print
The pair is
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]) ,
(!a(x).x[] | (^c)(!c[] | !c)))
> Check
Yes, size of the relation is 1
((^b)(!b.a(x).x[] | !a(t).t[] | !b[]),
(^c)(!c | !a(x).x[] | !c[]));
```

The syntax for processes is rather intuitive; we use `^` for restriction and square brackets for emission. Both processes are weakly bisimilar to $!a(x).\overline{x}$; here we use command `Switch` to toggle the bisimilarity checking mode (from strong to weak):

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

- Another law, which is a straightforward instantiation of the so–called *replication theorems*, that express the distributivity of private resources:

$$(\nu a)\,(!a(x).\overline{x} \mid !\overline{a}b \mid !\overline{a}c) \;\sim\; (\nu a)\,(!a(x).\overline{x} \mid !\overline{a}b) \mid (\nu a)\,(!a(x).\overline{x} \mid !\overline{a}c)$$

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:

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

Conclusion

We have developed some methods to automatically check bisimilarities between π–calculus processes, and shown their expressive power on a prototype implementation [2]. Our system is rather elementary, and cannot be compared as it is with other similar tools (like the Mobility Workbench [14], which is probably the closest to ours, Cesar/Aldebaran [4], the Jack Toolkit [2], or the

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

FC2tools package [1]). However, the possibility to reason on infinite states processes using the up to techniques is specific to our tool [3], and is an important feature in terms of expressiveness, as shown above.

Our system could be made more robust by enriching the syntax (e.g. adding definitions of agents and the choice operator), and improving the bisimulation checking method (this means in particular modifying our algorithm to adopt a breadth–first strategy, instead of making "blind" recursive calls to the general bisimulation checking function; this would insure some kind of "computational completeness": if a finite up–to bisimulation relation exists, we find it after a finite number of steps).

Completeness of our methods is a key theoretical issue, as it is directly related to the understanding of the expressiveness of our system. One would be interested in defining a class of terms (containing some infinite states processes) for which our algorithm is a decision procedure for bisimilarity. Let us explain informally where the difficulties come from: the key point is the up to parallel composition technique, that gives the possibility to reason with replicated terms. This technique is used to cancel common parallel components in two processes *right after* a transition has taken place. For example, take a process of the form $!a(\boldsymbol{b}).P$, liable to perform the transition $!a(\boldsymbol{b}).P \xrightarrow{a(\boldsymbol{c})} P' = !a(\boldsymbol{b}).P \mid P_{\boldsymbol{b}:=\boldsymbol{c}}$, and suppose $P \sim Q$, which implies $Q \xrightarrow{a(\boldsymbol{c})} Q'$: intuitively, the idea is that one should be able to cancel $P_{\boldsymbol{b}:=\boldsymbol{c}}$ *both* in P' and Q' (if we want to prove $P \sim Q$), otherwise P' and Q' could do the transitions $\xrightarrow{a(\boldsymbol{c'})}$, $\xrightarrow{a(\boldsymbol{c''})}$, ... , and the relation would keep growing *ad infinitum*. This phenomenon actually restricts considerably the freedom in the definition of the terms we can manipulate (hence the simplicity of the examples of Section 3), and it seems indeed that the up to parallel composition proof technique cannot be used as a brute force tool to prove bisimilarity results between infinite states processes. On the contrary, the idea is rather to use the automation of the up to techniques to *verify* a proof on paper, once we know that the up to techniques apply. Following this approach, work is in progress to adapt our methods to *open terms* [10,12], in order to be able to prove not only bisimilarity results, but also general bisimilarity laws (like for example the *replication theorems*).

Another interesting direction could be the mechanisation of the proofs of this paper, reusing the work of [5], which could allow one to extract a *certified* bisimilarity checker. Some more work has to be done in order to make these proofs tractable for the purpose of a theorem prover formalisation.

[3] We are aware of some efforts regarding the implementation of the up to techniques within the CONCUR project, but do not have details about the focus of this study and its outcome.

Acknowledgement

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

[1] A.Bouali, A.Ressouche, V.Roy, and R.de Simone. The FC2 Toolset. demo presentation at TACAS'96, AMAST'96 and CAV'96, 1996.

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

[3] J. Engelfriet and T. Gelsema. Structural Inclusion in the pi-Calculus with Replication. Report 98-06, Leiden University, 1998.

[4] J.-C. Fernandez, H. Garavel, A. Kerbrat, R. Mateescu, L. Mounier, and M. Sighireanu. Cadp (caesar/aldebaran development package): A protocol validation and verification toolbox. In *Proceedings of CAV' 96*, volume 1102 of *LNCS - Springer Verlag*, pages 437–440, 1996.

[5] 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.

[6] D. Hirschkoff. Automatically Proving Up-to Bisimulation. Technical Report 98-123, CERMICS, Champs sur Marne, March 1998.

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

[8] M. Pistore and D. Sangiorgi. A partition refinement algorithm for the π-calculus. In Rajeev Alur, editor, *Proceedings of CAV '96*, volume 1102 of *LNCS*, 1996.

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

[10] 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.

[11] 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.

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

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

[14] 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 *LNCS*, pages 428–440. Springer-Verlag, 1994.