Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Bisimulation Verification Using the Up-to Techniques

Checking Up-to Bisimulation, with an Application to the π-Calculus

Daniel Hirschkoff

LIP, ENS Lyon, 46, allée d'Italie, 69364 Lyon Cedex 07 - France
e-mail: `Daniel.Hirschkoff@ens-lyon.fr`

Received: date / Revised version: date

Abstract. This paper investigates the use of the so-called "up-to techniques" for bisimulation in the framework of verification. We first introduce a method to exploit these techniques, that originate in theoretical works, in the applied context of verification. We then apply it on the π-calculus, in order to design an up-to bisimulation verification algorithm for π-calculus terms. The outcome of such an effort is evaluated on two examples that were run on a prototype implementation.

1 Introduction

Motivation Recent evolutions in *how* and *what for* networks are used have put the stress on new questions in the field of study of concurrent systems. Issues like localities and failures, mobility, or security, rely on models of concurrency that depart from the usual modeling framework in the development of formal methods for the analysis of systems. For example, the notion of a fixed network topology over which communicating software can be run is not sensible the context of the world wide web or when running a program on a moving hardware device (see the distinction between *mobile computing* and *mobile computation* in [Car99], where a survey on these "complex forms of concurrency" is presented). Among other models, the π-calculus [Mil91] (together with related formalisms based on the paradigm of *name passing*) has emerged in the last decade as a simple and expressive algebra where such phenomena can be modeled. Indeed, stemming from the work on the π-calculus, related formalisms have succeeded in capturing notions such as locality [FG96, Ama97], mobility [CG98], or security [AG99]. Exploiting the π-calculus in a practical setting can be achieved in two ways. One can think of using it as a basis for the design of a *programming language* for concurrency. Implementations like Pict [PT98], JoCaml [FM97] or TyCo [Vas94] are related to this idea. Alternatively, one may want to use it as a *specification language*, with the aim of tool-supported analysis of π-calculus terms, the latter direction having been less investigated than the former.

One of the aims of the present work is to experiment in this field. More precisely, we focus here on *bisimilarity* [Par81, Mil89], which is probably the most popular equivalence when reasoning on π-calculus processes. This rather discriminating equivalence is based on the notion of *behaviour*. Bisimilarity properties are usually established using the *bisimulation* method, which in turn can be refined using powerful proof techniques. *Up-to techniques* are among these; [San95] gives a uniform presentation of the up-to techniques for bisimulation, in a general setting where a system is described using a Labelled Transition System. These techniques have turned out to be very helpful in allievating the burden needed by bisimulation proofs, in particular for the π-calculus and related languages, at least for "informal" proofs on the paper (see e.g. [MS92, San94, BS98, Bor98, RS99, BDNP99]).

Our goal in this work is to find out how the up-to techniques can be used to automatically verify bisimilarity properties, and what outcome one can expect from such an effort. Without entering yet the description of the up-to techniques, let us sketch the main lines of this work. We first set up a general framework for up-to bisimulation verification by adapting Mounier's "on the fly" checking algorithm. We then use these ideas to introduce a corresponding checking algorithm for the π-calculus. We focus in particular on the up to restriction and up to parallel composition proof techniques, which represent two powerful tools when reasoning about the π-calculus. The resulting verification algorithm has been implemented; this allows us then to illustrate its behaviour on two non-trivial examples, namely the treatment of persistent data in the π-calculus, and the verification of the alternating bit protocol.

The results presented in this paper first appeared in preliminary form in [Hir98,Hir99b].

Related Work Up-to techniques go back to Milner's bisimulation up to bisimilarity [Mil89]; up to context bisimulation has also appeared in a particular form in [Cau90], where the notion of *bisimulation base* is akin to what would correspond to bisimulation up to prefix. A more specific study of these techniques *per se* has been developped in [San95], on which most of the theoretical part of this work is based. Regarding up-to techniques for *weak* bisimilarity, [MS92] focuses on the problem of possible extensions of the bisimulation up to bisimilarity proof technique. To our knowledge, means to nicely extend the general theory of [San95] to the weak case are still to be found; an effort along these lines is pursued in [Fou98].

Regarding the problem of bisimulation verification, the standard method is probably the *partition refinement algorithm* [PT87,KS90]. However, as we shall see, Mounier's *on the fly* algorithm [FM91] will be more relevant to our setting. The former method is applied in [PS96] to check *open bisimulation* for π-calculus processes, and the latter is used in the Mobility Workbench [VM94] to reason about the fusion calculus (which can be seen as a dialect of π-calculus). It has to be noted that the Concurrency Workbench [CS96], as well as the Mobility Workbench, exploit a restricted form of bisimulation up to bisimilarity, by performing elementary garbage collections (such as cancellation of inactive processes) along the verification. Let us finally mention [FGM$^+$97], where a translation from π-calculus terms into *history-dependent automata* is defined, in order to reuse existing tools (namely FC2 and the Jack toolkit) to check for bisimulation. A common restriction in these studies is to work with *finite* π-calculus terms, where no parallel composition can occur under replication. In this work, however, we shall in some cases be able to reason about non-finite terms.

Outline The paper is organised as follows: we present the up-to techniques for bisimulation and a general method to perform verification using these in Section 2. We then turn to the π-calculus, which is briefly presented in Section 3, and introduce the corresponding up-to techniques (Section 4). Section 5 describes two case studies that were developped using a prototype implementation of our algorithm. We conclude and discuss future work in Section 6. The reader only interested in the general methodology (and not in its application to the π-calculus) can safely skip sections 3 to 5.

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

Fig. 2.1. Bisimilarity, Bisimulation, and Bisimulation up to $\mathcal{F}$

2 Up-to Bisimulation Verification

We introduce up-to bisimulation and design a general verification method. The results presented in this Section hold for any concurrent system which is specified as a *Labelled Transition System* (see below).

2.1 Bisimulation and Up-to Bisimulation

We first give a brief account on up-to bisimulation as presented in [San95].

Definition 1 (Labelled Transition System). A labelled transition system (LTS) is a triple ($\mathbb{P}$,$\mathbb{A}$,$\mathbb{T}$), where $\mathbb{P}$ is a set of states (or processes), ranged over by $P, Q \ldots$, $\mathbb{A}$ is a set of actions, ranged over by $\lambda, \mu, \ldots$, and $\mathbb{T}$ is a relation included in $\mathbb{P} \times \mathbb{A} \times \mathbb{P}$. We write $P \xrightarrow{\mu} P'$ for $(P, \mu, P') \in \mathbb{T}$.

Definition 2 (Bisimilarity). Given a LTS, bisimilarity, written $\sim$, is the greatest symmetric relation s.t. whenever $P \sim Q$ and $P \xrightarrow{\mu} P'$, there exists a state Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P' \sim Q'$.

Definition 2 can be represented on the left diagram of Figure 2.1; simple reasonings suffice to show that $\sim$ is an equivalence relation. To establish that two processes (or states) P and Q are bisimilar, the most popular method is *bisimulation*, depicted by the central diagram of Figure 2.1: exhibit a relation $\mathcal{R}$, relating P and Q such that the definitional property of $\sim$ holds for $\mathcal{R}$ (i.e. replace $P \sim Q$ and $P' \sim Q'$ by $P\mathcal{R}Q$ and $P'\mathcal{R}Q'$ respectively).

However, as it is the case for bisimilarity, the bisimulation proof method exhibits a form of circularity: a bisimulation must contain some pairs of processes and all possible derivatives (according to the "bisimulation game") of these processes, and so on … The idea behind the introduction of the up-to techniques for bisimulation [San95] is somehow to "refine" this circularity, by considering the proof schema on the right of Figure 2.1, where $\mathcal{F}$ is some function from relations to relations. Here, we have to establish that processes related by $\mathcal{R}$ evolve to processes related by $\mathcal{F}(\mathcal{R})$, instead of remaining within $\mathcal{R}$: intuitively, the rôle of $\mathcal{F}$ is to help constructing the "future" of a relation. This way, one can expect relieving the task of bisimulation proofs, by reducing the size of the relations to be examined. Let us formally state what it means to be a bisimulation up to $\mathcal{F}$:

Definition 3 (Up-to Bisimulation). Given a function $\mathcal{F}$ from relations to relations, a symmetric relation $\mathcal{R}$ is a bisimulation up to $\mathcal{F}$ if and only if whenever $P\,\mathcal{R}\,Q$ and $P \xrightarrow{\mu} P'$, there exists Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\,\mathcal{F}(\mathcal{R})\,Q'$.

Of course, not any function $\mathcal{F}$ on relations qualifies for the up-to bisimulation proof method to be correct in the quest for bisimilarity. In [San95], Sangiorgi calls *soundness* the property of such functions:

Definition 4 (Soundness). A function $\mathcal{F}$ is *sound* if and only if ($\mathcal{R}$ is a bisimulation up to $\mathcal{F}$) implies ($\mathcal{R} \subseteq \sim$).

When working with functions from relations to relations, we shall actually restrict ourselves to a subclass of the sound functions. This subclass is defined using the notion of *progressions of relations*:

Definition 5 (Progression of relations). We say that a relation $\mathcal{R}$ *progresses* to a relation $\mathcal{S}$, written $\mathcal{R} \rightarrow \mathcal{S}$, if and only if whenever $P\,\mathcal{R}\,Q$ and $P \xrightarrow{\mu} P'$, there exists Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P'\,\mathcal{S}\,Q'$, and symmetrically for the transitions of Q.

In particular, $\mathcal{R} \rightarrow \mathcal{F}(\mathcal{R})$ means that $\mathcal{R}$ is a bisimulation up to $\mathcal{F}$. We can now introduce *respectful* functions:

Definition 6 (Respectfulness). A function $\mathcal{F}$ is *respectful* if and only if, whenever $\mathcal{R} \subseteq \mathcal{S}$ and $\mathcal{R} \rightarrow \mathcal{S}$, then $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{S})$ and $\mathcal{F}(\mathcal{R}) \rightarrow \mathcal{F}(\mathcal{S})$.

A fundamental property of respectfulness is that it is preserved by function composition, which will allow us to combine several up-to techniques if necessary. The following property shows that respectful functions are good candidates for up-to techniques:

Theorem 2.1 [San95]. *A respectful function is sound.*

The notion of up-to bisimulation can also be introduced in the *weak case*, where the set of actions $\mathbb{A}$ contains a special action τ, considered as unobservable. Several notions of equivalence and preorder can then be considered. We give below the definitions of weak bisimilarity and expansion (which will be used in paragraph 5.2):

Definition 7 (The Weak Case). We write $\Rightarrow$ for the reflexive transitive closure of $\xrightarrow{\tau}$, $\overset{\mu}{\Rightarrow}$ for $\Rightarrow \xrightarrow{\mu} \Rightarrow$, and $\xrightarrow{\hat{\mu}}$ stands for $\xrightarrow{\mu}$ if $\mu \neq \tau$, and for $=$ or $\xrightarrow{\tau}$ if $\mu = \tau$.

Weak bisimilarity ($\approx$) is the greatest relation such that whenever $P \approx Q$ and $P \xrightarrow{\mu} P'$, there exists Q' s.t. $Q \overset{\hat{\mu}}{\Rightarrow} Q'$ and $P' \approx Q'$, and symmetrically for the transitions of Q.

Expansion, written $\lesssim$, is the greatest relation such that, whenever $P \lesssim Q$, $P \xrightarrow{\mu} P'$ implies that there exists a Q' s.t. $Q \overset{\mu}{\Rightarrow} Q'$ and $P' \lesssim Q'$, and $Q \xrightarrow{\mu} Q'$ implies that there exists P' s.t. $P \xrightarrow{\hat{\mu}} P'$ and $P' \lesssim Q'$.

Intuitively, $\overset{\mu}{\Rightarrow}$ transitions are to be interpreted as transitions along μ in which some extra internal computation is performed. It holds that $\approx$ is an equivalence relation, and $\lesssim$ is a preorder. Expansion is of interest when examining specifications, as it allows one to express the fact that two systems have the same behaviour modulo extra invisble transitions (typically, the implementation behaves like the specification *plus* some silent steps). As in the strong case, we introduce the notions of weak bisimulation and expansion, along with their corresponding up-to versions.

We conclude this succint presentation of the framework of up-to bisimulation by revisiting Milner's *bisimulation up to bisimilarity* [Mil89] in this setting:

Example 2.2 (Bisimulation up to bisimilarity). Let

$$\mathcal{F}_{\sim}(\mathcal{R}) \overset{def}{=} \{(P,Q).\exists(P_0,Q_0) \in \mathcal{R}.\, P \sim P_0 \,\wedge\, Q_0 \sim Q\}.$$

Suppose that $\mathcal{R}$ is a bisimulation up to $\mathcal{F}_{\sim}$. Then $\mathcal{F}_{\sim}(\mathcal{R})$ is a bisimulation (simple diagram chasing, using transitivity of $\sim$). Hence, since $\mathcal{R} \subseteq \mathcal{F}_{\sim}(\mathcal{R}) \subseteq \sim$, $\mathcal{F}_{\sim}$ is sound. In practice, this allows one, when "playing the bisimulation game", to replace processes resulting from a transition by terms which are known to be bisimilar.

We shall see that in some cases, and in particular in the π-calculus, rather elaborate sound functions can be used in order to reduce the burden needed by bisimulation proofs. As a matter of fact, an important feature of the functions we will work with is that their definition relies on the syntactical description of the states: this is actually crucial in the design of a bisimulation verification procedure, which we examine now.

2.2 The Verification Method

Classical methods for bisimulation verification take as input the state spaces which can be computed from the initial processes to be compared. The first step in this setting is thus to unfold all possible evolutions from the initial states, and store the resulting transition systems using appropriate implementation techniques such as *binary decision diagrams* [Bry94]. Then, the most commonly used method to check if a bisimulation relation relating the state spaces can be found is the *partition refinement algorithm* [PT87,KS90]. Such an approach has been developped in [PS96] to check *finite* π-calculus processes.

We do not use here this technique, for two main reasons. The first reason is that, as we said above, the functions corresponding to the up-to techniques we shall be working with perform *syntactical* manipulations on terms: we therefore need to have access to the actual representation of the states along the execution of a process, instead of getting rid of it as is usually done when computing the state space. The second, fundamental reason, is that our proof techniques for the π-calculus will sometimes make it possible to perform *finite* proofs involving processes whose state space is actually *infinite*: in such cases, we do not want to enter the computation of the state spaces, since that would directly throw us into an infinite loop.

We are therefore led to take a more intensional point of view and adopt Mounier's *on the fly* checking method for bisimulation [FM91]. The overall principle of this approach is to refer to the definition of bisimulation and try to incrementally build a bisimulation relating the two initial states. Working within this framework will allow us to exploit the algebraic reasoning associated to our up-to techniques. The on the fly checking algorithm explores the *state spaces product* of the two processes to be compared, defined as follows:

Definition 8 (State Spaces Product). The state spaces product of two processes P_{init} and Q_{init} is the transition system generated from the initial state (P_{init}, Q_{init}) using the following inference rule:

$$\frac{\{P \xrightarrow{\mu_i} P'_i\}_{i\in I} \quad \{Q \xrightarrow{\lambda_j} Q'_j\}_{j\in J} \qquad \{\mu_i\}_{i\in I} = \{\lambda_j\}_{j\in J} \quad \mu_{i_0} = \lambda_{j_0}}{(P,Q) \xrightarrow{\mu_{i_0}} (P'_{i_0}, Q'_{j_0})}.$$

The rule above reads as follows: we want the state (P,Q) to have successor states only if processes P and Q exhibit the same sets of actions in their possible transitions (condition $\{\mu_i\}_{i\in I} = \{\lambda_j\}_{j\in J}$). When this holds, the successor states of (P,Q) are all the (P'_{i_0}, Q'_{j_0}) such that P'_{i_0} and Q'_{j_0} correspond to evolvings of P and Q respectively along the same action μ_{i_0}.

To check for bisimulation, we want to find a subset of the state spaces product containing the initial pair and that is closed w.r.t. the associated transition relation. In trying to incrementally construct this set, the core of the procedure consists in examining a pair of processes (P,Q), and in distinguishing between three cases:

1. Either we can give an immediate diagnosis: $P \sim Q$ (e.g. if $P\mathcal{R}_0Q$ for some decidable subset $\mathcal{R}_0$ of $\sim$), or $P \nsim Q$ (if P and Q exhibit different immediate transitions),
2. Or the pair has not been examined yet during the checking process, in which case make a bet: suppose that $P \sim Q$, and recursively inspect the sons of (P,Q),
3. Or the pair has already been met and we have supposed it to consist of bisimilar terms, in which case the definitional property of bisimulation holds: inspect other pairs (if any).

Case 2 above corresponds to the point where the phenomenon of *state explosion*, which is typical of concurrent systems, may occur.

Let us now define more precisely an algorithm to implement the ideas above (we refer to Figure 2.2). The data structures we work with are: a structure $\mathbb{S}$ (we shall return later on the nature of $\mathbb{S}$), that contains the states that still have to be examined; three sets V, W, and R, containing the pairs of processes that are respectively supposed to be bisimilar, known to be non-bisimilar, and known to be bisimilar. When two new processes turn out to be bisimilar, if the corresponding pair is in V, it is stored in R, and symmetrically for the case of non-bisimilarity (from V to W). It can be the case, though, that during the exploration of the state space, an incorrect hypothesis is assumed regarding the bisimilarity of two processes (i.e. it turns out that a pair in R is actually made of non-bisimilar processes): to handle this situation, a boolean flag called `status` records the "reliability" of the current computation. When a computation leads to an unreliable result, we start all over again, keeping track in W of the states that have been found non bisimilar. The main algorithm (given by procedure **bisimulation_check**) uses three auxiliary functions: **succeeds**, **fails**, and **propagate** (whose arguments are omitted in the main algorithm for the sake of readability). **succeds** and **fails** test for an immediate diagnosis (see point 1 above); while **succeeds** does not rely on R and V, **fails** also uses the information contained in W to check for immediate failure. Function **propagate** is central to the algorithm: its rôle is to update the known information regarding the state spaces product, possibly discovering new (non-) bisimilarity properties, or cutting out branches that do not have to be explored anymore (it can be the case that the bisimulation property has been established for a given pair of terms before all possible pairings in its descendants have been examined). The arguments of **propagate** are: the two processes P and Q for which an information has just been found, and a boolean value `success` telling whether P and Q have been found bisimilar or not. Depending on the place in the main loop where **propagate** is called, `success` is equal to false (only in the case "(P_0, Q_0) **fails**"), or to true (in all other calls to **propagate**). In the case where a negative information is in contradiction with the hypotheses that have been used along execution (as recorded in set R), **propagate** finds a "wrong bet", and sets the global variable `status` to false.

One of the advantages of the *on the fly* checking method is that, by performing verification along the construction of the state space, we can hope to reach a bisimulation before the whole state space has been computed, thus reducing the memory requirements. As expected, the nature of the structure $\mathbb{S}$ determines the exploring strategy: choosing a *stack* (resp. *queue*) induces a depth-first (resp. breadth-first) strategy. Depending on the behaviour of the systems we examine, either choice can lead to more efficient results. It has to be stressed, however, that the depth-first strategy is more adapted to handle negative responses of the algorithm: indeed,

bisimulation_check(P, Q) :=
W:= $\emptyset$;
$(*)$ R:= $\emptyset$, V:= $\emptyset$, insert (P, Q) in $\mathbb{S}$, `status`:= `true`
`while` $\mathbb{S} \neq \emptyset$ `do`
  take (P_0, Q_0) in $\mathbb{S}$;
  `if` (P_0, Q_0) **succeeds** `then` **propagate** `else`
  `if` (P_0, Q_0) **fails**
   `then` (insert (P_0, Q_0) in W; **propagate**) `else`
  `if` $\boxed{(P_0, Q_0) \in \mathrm{V}}$
   `then` (bring (P_0, Q_0) from V to R; **propagate**) `else`
  `if` $\boxed{(P_0, Q_0) \in \mathrm{R}}$ `then` **propagate** `else`
  `if` $(P_0, Q_0) \in \mathrm{W}$ `then` **propagate** `else`
   {- *pair* (P_0, Q_0) *is new* -}
  store the successors of (P_0, Q_0) in $\mathbb{S}$
`done`
`if` $P \not\sim Q$ `then` return `false`
  `else if status then` return `true else` loop back to $(*)$

succeeds(P, Q) :=
  $(P, Q) \in \mathcal{R}_0$, where $\mathcal{R}_0$ is a decidable subset of $\sim$
   (e.g. $\equiv$ in the π-calculus – see below)

propagate$(P, Q$,success) :=
  let (P_f, Q_f) be the father of (P, Q) in the state space
   (we have $(P_f, Q_f) \xrightarrow{\mu} (P, Q)$ for some μ);
  `if success`
  `then` {- success = true -}
   remove from $\mathbb{S}$ all sons of (P_f, Q_f) along transition μ
   `if` $\forall \mu'. \nexists (P', Q') \in \mathbb{S}$ s.t. $(P_f, Q_f) \xrightarrow{\mu'} (P', Q')$
    `then` {- *all sons of* (P_f, Q_f) *have been inspected* -}
     `if` (P_f, Q_f) is in V, store it in R;
     propagate$(P_f, Q_f$,true);
  `else` {- success = false -}
   `if` $(P, Q) \in \mathrm{R}$ then `status` := false;
   `if` $\nexists (P', Q') \in \mathbb{S}$ s.t. $(P_f, Q_f) \xrightarrow{\mu} (P', Q')$ `then`
    {- *we can conclude that* $P_f \not\sim Q_f$ -}
     add (P_f, Q_f) to W;
     propagate$(P_f, Q_f$,false)

fails(P, Q) :=
  $P \xrightarrow{\mu} P'$ and $\forall Q'. Q \not\xrightarrow{\mu} Q'$, or the symmetrical case,
  or $(P, Q) \in \mathrm{W}$

Fig. 2.2. "On the Fly" Bisimulation Checking Algorithm and auxiliary functions

since it follows closely the transitions, it makes it easier to exhibit runs that show the non-bisimilarity property. The up-to techniques we shall define in Section 4 allow one to reason in some cases about *infinite* behaviours, by matching two infinite branches in the state spaces, and cutting them. The issue of finding the right matches between infinite branches is thus crucial, as this may turn an infinite proof into a finite one. We therefore adapt Mounier's original algorithm, that uses a depth-first strategy, to breadth-first, thus obtaining a *computationally complete* checking algorithm:

Definition 9 (Computational Completeness). We say that a bisimulation checking algorithm is *computationally complete* with respect to a given up-to technique $\mathcal{F}$, whenever it holds that the algorithm diverges if and only if no finite bisimulation up to $\mathcal{F}$, relating the two processes to be checked and derivatives of these processes, exists.

Theorem 2.3. *The breadth-first version of the algorithm of Figure 2.2 is computationally complete.*

The proof of this result follows immediately from the very definition of breadth-first search. Not much has to be changed to transform this on the fly bisimulation checking algorithm in order to handle up-to bisimulation. Indeed, by referring to the schemas of Figure 2.1, we can see that we only need to replace in procedure **bisimulation_check** $\boxed{(P_0, Q_0) \in \mathrm{V}}$ and $\boxed{(P_0, Q_0) \in \mathrm{R}}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(\mathrm{V})}$ and $\boxed{(P_0, Q_0) \in \mathcal{F}(\mathrm{R})}$ respectively, $\mathcal{F}$ being our up-to technique (since sets R and V are used to embody the "bisimulation game" during the run of the algorithm).

Important remark: as we perform this modification, we *do not* change the test $(P_0, Q_0) \in \mathrm{W}$, since W contains pairs that should *not* belong to the up-to bisimulation relation. Note as well that in the case where we have to insert new pairs into $\mathbb{S}$, we insert the successors of (P_0, Q_0) *as they are*, i.e. without using function $\mathcal{F}$ to possibly simplify the pair.

In order to perform the modified tests involving $\mathcal{F}(\mathcal{R})$, we further need to establish a decidability property for statements of the form $(P, Q) \in \mathcal{F}(\mathcal{R})$, which often amounts to inverting $\mathcal{F}$ (as will be seen below).

Summary of the Up-to Bisimulation Verification Method
Let us recall the ingredients that are needed in order to design an up-to bisimulation checking method:

- We use a *Labelled Transitions System* to specify systems; this induces an associated notion of bisimulation between states.
- Up-to techniques are given by sound functions over relations between states.
- To apply the "on the fly" method, we need to be able, for every such function $\mathcal{F}$, given two states P and Q and a relation $\mathcal{R}$, to decide whether $(P, Q) \in \mathcal{F}(\mathcal{R})$.

In order to evaluate the outcomes of using the up-to techniques for verification, we apply this general framework to the π-calculus: we first provide instantiations for the three points listed above, then run an implementation of the corresponding algorithm on a few examples.

3 The π-Calculus

In this Section, we introduce the syntax and semantics of the language we use, a restricted though expressive subset of polyadic π-calculus where replication is allowed only on prefixed processes (see the conclusion for remarks on possible extensions of the language). We then define bisimulation, together with up-to techniques.

Definition 10 (Syntax). Given an infinite countable set of *names* $\mathcal{N}$, ranged over by $a, b, c, \ldots$, we range over (possibly empty) name lists with $\widetilde{a}, \widetilde{b}, \ldots$, and we define *prefixes*, ranged over by α, β, as follows:

$$\alpha \stackrel{def}{=} a(\widetilde{b}) \mid \overline{a}[\widetilde{b}] \,;$$

Processes, ranged over by $P, Q, R, \ldots$, are then given by the syntax:

$$P \stackrel{def}{=} \mathbf{0} \mid (\alpha.P)^e \mid (\nu x)\, P \mid P_1|P_2 \,,$$

where $e \in \mathbb{N} \cup \{\omega\}$.

Prefixed processes are either receptions (prefix $a(\widetilde{b})$) or emissions ($\overline{a}[\widetilde{b}]$); they come with a *multiplicity* e, denoting the number of copies of the prefixed processes that are put in parallel (multiplicity ω stands for arbitrarily many copies). $\mathbf{0}$ is the inactive process; the restriction operator ν makes a name private to the restricted process; parallel composition is written $|$.

As usual, the *free names* of a process P, written $\text{fn}(P)$, are defined by saying that reception and restriction are binding constructs.

Notations. To recover the usual π-calculus notation for replication, we let $!\alpha.P$ be a synonym for $(\alpha.P)^\omega$. In prefixes $a(\widetilde{b})$ and $\overline{a}[\widetilde{b}]$, a and $\widetilde{b}$ are called respectively the *subject* and the *object* parts of the prefix. We shall omit the object part of a prefix when it is empty, and use a monadic notation for single name object parts (thus writing e.g. $a(b)$ and $\overline{a}b$). We shall as well omit the continuation of a prefix when it is the inactive process $\mathbf{0}$. $\prod_{i=1}^k P_i$ stands for $(P_1|\ldots|P_k)$. Free and bound names are defined by saying that restriction and abstraction (embodied in the input prefix) are binding operators. Simultaneously substitution of name list $\widetilde{a}$ by $\widetilde{b}$ in a process P is written $P_{\widetilde{a}:=\widetilde{b}}$. As usual, we shall silently use α-conversion to avoid capture of bound names.

Definition 11 (Transition System). We first introduce the notation $\tilde{\tilde{x}}$, that stands for a (possibly empty) *set* of names; given two disjoints sets $\tilde{\tilde{x}}$ and $\tilde{\tilde{y}}$, we write $\tilde{\tilde{x}} \uplus \tilde{\tilde{y}}$ to denote their union.

The operational semantics of π-calculus terms is defined as a labeled transition system. *Actions*, ranged over by μ, μ', are given by the following syntax:

$$\mu \stackrel{def}{=} a(\widetilde{b}) \mid (\nu\tilde{\tilde{b'}})\, \overline{a}[\widetilde{b}]_{\tilde{\tilde{b'}}\subseteq\widetilde{b}} \mid \tau \,,$$

which reads as follows: an action is either a *reception*, a (possibly bound) output, or the silent action τ, denoting internal communication. A bound output $\overline{a}[\widetilde{b}]_{\tilde{\tilde{b'}}\subseteq\widetilde{b}}$ represents the emission of a *list* of names $\widetilde{b}$ along channel a, among which names belonging to the *set* $\tilde{\tilde{b'}}$ are private to the sender. Bound and free names of actions are defined as usual, and we use the notation $n(\mu)$ for the names (bound and free) occurring in an action μ.

The judgment $P \xrightarrow{\mu} P'$, meaning that process P is liable to perform action μ to become P', is defined by the rules of Figure 3.1 (symmetrical versions of rules PAR_l and CLOSE_1 are omitted). In rule (OPEN), $\widetilde{b}\backslash\tilde{\tilde{b'}}$ represents the list obtained by erasing the names occurring in the set $\tilde{\tilde{b'}}$ from the list $\widetilde{b}$. Note that we adopt an *early* version of the operational semantics (*late* and *open* presentation also exist).

Terminology and convention. In defining our up-to bisimulation checking algorithm for the π-calculus, we shall restrict ourselves to relations between terms consisting in a *finite* number of pairs of processes (this excludes e.g. definitions like $\mathcal{R} \stackrel{def}{=} \{(\mathbf{0}, \overbrace{\mathbf{0}|\ldots|\mathbf{0}}^{n \text{ times}})\}$). In the following, we shall refer to such objects simply as ***relations***.

4 Up-to Techniques for the π-calculus

There are four up-to techniques that are commonly used to reason about π-calculus processes, namely the up to bisimilarity, up to injective substitutions, up to restriction and up to parallel composition proof techniques. We establish here some decidability results that will allow us to apply the general method from paragraph 2.2 in order to verify up-to bisimulation for the π-calculus.

Several proofs in this Section involve some rather technical reasoning on π-calculus terms, which may be skipped by a reader interested only in the application of the up-to techniques for verification. If so, the relevant results are Theorems 4.2, 4.7 and 4.9 (see also Defs. 12, 15 and 16).

4.1 Up to Structural Congruence

Structural Congruence We first want to have a reasonable notion of *state* when comparing the behaviour of π-calculus processes. This is achieved by somehow hiding a few elementary syntactic manipulations on terms, which merely express the basic properties of the constructors of the calculus. Such properties are traditionally regrouped in the *structural congruence relation*, defined as follows:

Definition 12 (Structural Congruence). (Extended) structural congruence, written $\equiv$, is the least equivalence relation that is a congruence for all π-calculus operators and that satisfies the rules of Figure 4.1.

$$(\text{INP})\ a(\widetilde{b}).P \xrightarrow{a(\widetilde{c})} P_{\widetilde{b}:=\widetilde{c}} \qquad (\text{OUT})\ \overline{a}[\widetilde{b}].P \xrightarrow{\overline{a}[\widetilde{b}]} P \qquad (\text{BANG})\ \frac{\alpha.P \xrightarrow{\mu} P'}{(\alpha.P)^\omega \xrightarrow{\mu} (\alpha.P)^\omega|P'}$$

$$(\text{RES})\ \frac{P \xrightarrow{\mu} P'}{(\nu x)\,P \xrightarrow{\mu} (\nu x)\,P'}\ x \notin n(\mu) \qquad (\text{PAR}_l)\ \frac{P \xrightarrow{\mu} P'}{P|Q \xrightarrow{\mu} P'|Q}\ bn(\mu) \cap fn(Q) = \emptyset$$

$$(\text{OPEN})\ \frac{P \xrightarrow{(\nu\tilde{\tilde{b}}')\,\overline{a}[\widetilde{b}]} P'}{(\nu t)\,P \xrightarrow{(\nu\{t\}\uplus\tilde{\tilde{b}}')\,\overline{a}[\widetilde{b}]} P'}\ t \neq a, t \in \widetilde{b} \setminus \tilde{\tilde{b}}' \qquad (\text{CLOSE}_1)\ \frac{P \xrightarrow{a(\widetilde{b})} P' \quad Q \xrightarrow{(\nu\tilde{\tilde{b}}')\,\overline{a}[\widetilde{b}]} Q'}{P|Q \xrightarrow{\tau} (\nu\tilde{\tilde{b}}')\,(P'|Q')}\ \tilde{\tilde{b}}' \cap fn(P) = \emptyset$$

Fig. 3.1. Operational Semantics

R1	$P\|Q \equiv Q\|P$	
R2	$P\|(Q\|R) \equiv (P\|Q)\|R$	
R3	$(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$	
R4	$(\nu x)P \mid Q \equiv (\nu x)(P\|Q)$	if $x \notin \mathrm{fn}(Q)$
R5	$P \mid \mathbf{0} \equiv P$	
R6	$(\nu x)\,\mathbf{0} \equiv \mathbf{0}$	
R7	$(\nu x)\,(\alpha.P)^e \equiv \mathbf{0}$	if $x = \mathrm{subject}(\alpha)$
R8	$(\alpha.P)^{e_1} \mid (\alpha.P)^{e_2} \equiv (\alpha.P)^{e_1 \oplus e_2}$	

$$\text{where } e_1 \oplus e_2 = \begin{cases} e_1 + e_2 \text{ if } (e_1, e_2) \in \mathbb{N}^2 \\ \omega \text{ otherwise} \end{cases}$$

Fig. 4.1. Structural Congruence

We refer to $\equiv$ as *extended* structural congruence because of rule R7, which is not included in traditional presentations [Mil92, Mil91]; we shall return to this point in paragraph 5.2. The laws of structural congruence can be derived as bisimilarity results, i.e. it can be proved that $\equiv \subseteq \sim$. Moreover, we are interested in establishing the decidability of $\equiv$, in order to define a bisimulation up to structural congruence proof method. To do this, we first remark that by rules R1 and R2, parallel composition is an *Associative-Commutative* (AC) operator. Accordingly, we shall identify two terms that differ by applying only those two rules. We want to define a Term Rewrite System (TRS) on such processes; to do this, we first introduce an alternative formulation of rules R6 and R7, which will be useful for our purposes:

R6' $\quad (\nu x)\,P \;\equiv\; P$ if $x \notin \mathrm{fn}(P)$

R7' $\quad (\nu x)(\nu\widetilde{y}))\,((\alpha.P)^e \mid Q) \;\equiv\; (\nu\widetilde{y})\,Q$ if $x = \mathrm{subject}(\alpha)$ and $x \notin \mathrm{fn}(Q)$

We remark that R7' is a proof *schema* rather than a single rule, because of the implicit quantification over the size of $\widetilde{y}$. It is easily proved that the relation defined by rules R1 to R8 coincides with the relation where we replace R6 and R7 by the above definitions.

We are now ready to introduce our rewrite system:

Definition 13 (Term Rewrite System). We let $\Re$ be the Term Rewrite System, defined on π-calculus terms identified modulo α-conversion and AC properties of parallel composition, obtained by orienting rules R4, R5, R6', R7' and R8 from left to right.

Note that we leave out rule R3 for the moment (as well as R1 and R2, since we work up to AC of parallel composition). We write $A \to B$ whenever some subterm of A can be rewritten using one of the rules of $\Re$, yielding B; $\to^*$ stands for the reflexive transitive closure of $\to$. Due to our restricted syntax for terms, it is quite easy to establish that $\Re$ enjoys the uniqueness of normal forms property[1]:

Proposition 4.1 (Uniqueness of Normal Forms). *$\Re$ is strongly normalising and confluent.*

Proof.

Local confluence: we have to show that whenever $A \to B$ and $A \to C$, there exists D such that $B \to^* D$ and $C \to^* D$. This is proved by examining the *critical pairs* associated to $\Re$; these correspond to possible conflicts in the application of the rewrite rules (see example below). In presence of an AC operator like parallel composition, though, we have to adapt the standard methodology, and consider, for every rewrite rule of the form $A|B \to C$, the *extended* rule $A|B|D \to C|D$ (see [PS81]; note that the "flat" notation $A|B|D$ is also a consequence of $|$ satisfying AC).

Examination of the induced critical pairs is then rather straightforward. Here is an example, resulting from the superposition of rules R4' (the extended version of R4) and R8. The term

$$(\nu x)\,P \mid (\alpha.Q)^{e_1} \mid (\alpha.Q)^{e_2}\,,$$

where we suppose that $x \notin \mathrm{fn}(\alpha.Q)$, can be rewritten into both

$$(\nu x)\,(P|(\alpha.Q)^{e_1}) \mid (\alpha.Q)^{e_2} \quad \text{and} \quad (\nu x)\,P \mid (\alpha.Q)^{e_1 \oplus e_2}\,.$$

[1] When replication is allowed on non-prefixed terms, the definition of structural congruence has to be extended – see [Hir99a].

We can then apply rules R4 then R8, and R4, respectively, to rewrite both terms into

$$(\nu x)\,(P \mid (\alpha.Q)^{e_1 \oplus e_2}).$$

Strong normalisation: to show that there is no infinite rewriting sequence, we introduce an order on terms which is the lexical product between two orders defined by measures on terms. The first measure is given by the number of symbols that are needed to write the term, while the second one is equal, for each ν occurring in the term, to the number of occurrences of $|$ surrounding this construct. The second measure is defined to handle rule R4, application of other rules being captured by the first measure. Both orders are well-founded, and thus define in turn a well-founded order on processes. Finally, since rewriting makes a term strictly decrease for this order, we can conclude that $\Re$ is strongly normalising.

Newmann's lemma [New42] then guarantees, given that the system is strongly normalising and confluent, that it is confluent. This entails unicity of normal forms for $\Re$, which means that two terms that are equated by laws R4, R5, R6', R7' and R8 yield by $\Re$ equal terms up to rules R1, R2, R3 and α-conversion. □

In the following, we shall denote by $\Re(P)$ the result of applying the above term rewrite system to P. We can now state the desired result:

Theorem 4.2. *It is decidable, given two processes P and Q, whether $P \equiv Q$.*

Proof. We write $=_{\alpha\nu|}$ to indicate that two terms are equal modulo α-conversion, AC properties of parallel composition, and permutation of consecutive restrictions (rule R3).

We show that $P \equiv Q$ implies $\Re(P) =_{\alpha\nu|} \Re(Q)$ by induction over the derivation of $P \equiv Q$. Since R3 is not involved in the definition of $\Re$, the only interesting case is when $P = (\nu x)(\nu y)\, P_0$ and $Q = (\nu y)(\nu x)\, P_0$. By confluence of $\Re$ (Proposition 4.1), we have that

$$\Re(P) = \Re(\,(\nu x)\Re(\,(\nu y)\,\Re(P_0)))\,,$$

and similarly for Q (indeed, confluence allows us to apply whatever strategy we want when computing $\Re(P)$). Now by inspection of the rewrite rules, $(\nu y)\,\Re(P_0)$ can only be rewritten using R6' or R7'. In both cases, it holds that if $(\nu y)\,\Re(P_0)$ can be rewritten, so does $(\nu y)(\nu x)\,\Re(Q_0)$ (which comes from the normalisation of Q), and that both resulting processes are equal (up to α conversion and AC). When no rewrite rule can be triggered, we apply rule R3 to show that $\Re(P) =_{\alpha\nu|} \Re(Q)$.

The other cases for the induction are straightforward, since they are encompassed by $\Re$. Moreover, it also holds that $\Re(P) =_{\alpha\nu|} \Re(Q)$ implies $P \equiv Q$ (by definition of $\Re$ and of $\equiv$). We thus have that for all processes P and Q, $P \equiv Q$ if and only if $\Re(P) =_{\alpha\nu|} \Re(Q)$. Therefore, to check if two processes are related by $\equiv$, just normalise them using $\Re$, and compare the resulting terms: since there may be only a finite number of permutations of restrictions in the terms, we can decide whether $A =_{\alpha\nu|} B$ for two arbitrary terms A and B, which concludes the proof. □

Notation. In the following, we shall identify processes that are equal modulo $=_{\alpha\nu|}$, and simply use the symbol $=$. Accordingly, consecutive restrictions have a set rather than a vector structure (rule R3). We therefore write $(\nu\tilde{\tilde{x}})\,P$ instead of $(\nu\widetilde{x})\,P$, to stress this point.

Normal Forms As will be seen below, Theorem 4.2 gives us a method to check for bisimulation up to structural congruence. In order to define the algorithms corresponding to the other proof techniques we want to apply, we first give a syntactical characterisation of normal forms.

Lemma 4.3 (Description of Normal Forms). *For any term P, $\Re(P)$ is either the inactive process $\mathbf{0}$, or a term of the form $(\nu\tilde{\tilde{x}})\,((\alpha_1.P_1)^{e_1}|\dots|(\alpha_k.P_k)^{e_k})$, where the P_is are normal processes such that the following conditions hold:*
(i) $(a \in \tilde{\tilde{x}}) \;\Rightarrow\; a \in \mathrm{fn}(\prod_i \alpha_i.P_i)$;
(ii) $(i \neq j) \;\Rightarrow\; (\alpha_i, P_i) \neq (\alpha_j, P_j)$;
(iii) $(a = subject(\alpha_i)) \;\Rightarrow\; a \notin \tilde{\tilde{x}} \,\vee\, a \in \mathrm{fn}(\prod_{j \neq i} \alpha_j.P_j)$.

Proof. Let $\mathcal{M}$ be the set of π-calculus processes which obey the syntax above but do not necessarely verify conditions (i), (ii) and (iii) in the second case. We first observe that whenever a term is in $\mathcal{M}$, all possible rewritings of this term are again in $\mathcal{M}$ (by examining the rewrite rules). Since the non-validity of criterion (i), (ii) or (iii) entails the applicability of rule R6', R8 or R7' respectively, we have by strong normalisation that every term in $\mathcal{M}$ gets finally rewritten into a term for which (i), (ii) and (iii) hold.

We then prove by induction on P that $\Re(P) \in \mathcal{M}$: the $\mathbf{0}$ and $(\alpha.P)^e$ cases are obvious. For $P = (\nu x)\,P_0$, the non-trivial case is when $\Re(P_0) \neq \mathbf{0}$, for which the property holds. For $P = P_1|P_2$, we repeatedly apply rule R4 after normalisation of P_1 and P_2 (and possibly rule R5) to obtain a process in $\mathcal{M}$.

We finally have that for any P, $\Re(P) \in \mathcal{M}$, and for any $T \in \mathcal{M}$, $\Re(T)$ is described by the syntax above, hence any term is rewritten into a term obeying this syntax. Reciprocally, it is clear that no such term can be rewritten, so we indeed have a description of the normal forms w.r.t. $\Re$. □

Non-trivial processes in normal form are thus built from a parallel composition of prefixed terms, on top of which some restrictions may be found. This intuition leads to the introduction of the following notations:

Notation. We decompose a process P in normal form of the shape $(\nu\tilde{\tilde{x}})\,((\alpha_1.P_1)^{e_1}|\dots|(\alpha_k.P_k)^{e_k})$ into its *body* $((\alpha_1.P_1)^{e_1}|\dots|(\alpha_k.P_k)^{e_k})$, written $\langle P\rangle$, and its *topmost* *restrictions*, that we write $\nu\tilde{\tilde{x}}_P$ to indicate the dependency on P. We extend this notation to the inactive process by letting $\nu\tilde{\tilde{x}}_{\mathbf{0}} \stackrel{def}{=} \nu\emptyset$ and $\langle\mathbf{0}\rangle \stackrel{def}{=} \mathbf{0}$.

Closure up to Structural Congruence We can now define our first up-to technique to reason with π-calculus terms:

Definition 14 ($\equiv\_\equiv$ function).

$$\equiv\mathcal{R}\equiv \stackrel{def}{=} \{(P,Q).\, \exists P_0, Q_0.\, P \equiv P_0 \mathcal{R} Q_0 \equiv Q\}\,.$$

Along the lines of Example 2.2, we get:

Proposition 4.4. *$\equiv\_\equiv$ is a respectful function.*

Corollary 4.5 (of Theorem 4.2). *Given two processes P, Q and a decidable relation $\mathcal{R}$, it is decidable whether $(P,Q) \in \equiv\mathcal{R}\equiv$.*

In the remainder of this Section, we shall systematically reason about processes in normal form: every relation $\mathcal{R}$ we shall manipulate will be supposed to be made of normalised terms, and we will compose every function $\mathcal{F}$ over relations with $\equiv\_\equiv$, by associating $\equiv\mathcal{F}(\mathcal{R})\equiv$ to $\mathcal{R}$. Concretely, in playing the bisimulation game, this amounts to applying the normalisation procedure immediately after every transition.

4.2 Up to Injective Substitutions and Restrictions

We now introduce the up to injective substitutions and restrictions proof technique. In particular, working up to restrictions is useful to handle the *name extrusion* mechanism of the π-calculus, which is at the heart of the expressiveness of this formalism.

Definition 15 ($\_^{i\nu}$ function). The function on relations corresponding to the up to injective substitutions and up to restrictions proof technique, written $\_^{i\nu}$, is defined as follows:

$$\mathcal{R}^{i\nu} \stackrel{def}{=} \{(\,(\nu\tilde{\tilde{x}})\,(P\sigma),\,(\nu\tilde{\tilde{x}})\,(Q\sigma)\,).\; P\,\mathcal{R}\,Q \text{ and } \sigma \text{ is injective on } \mathrm{fn}(P|Q)\}\,.$$

Proposition 4.6 [San95]. *$\_^{i\nu}$ is a respectful function.*

Proof (Sketch). This result, as well as Proposition 4.8 below, is a consequence of Proposition 5.2 of [San95], which states the respectfulness of the function that computes the closure under contexts of a given relation. Contexts are π-calculus terms containing a *hole*, written []. We compute this function on a given relation $\mathcal{R}$ by adding, for every pair of terms (P,Q) in $\mathcal{R}$, the pair $(C[P], C[R])$, where $C[T]$ is the term obtained by filling the hole in C with T, for every context C. The proof of respectfulness is then an induction over the structure of *contexts.* It can be adapted to our setting without much effort, to prove respectfulness of both the up to restriction and up to parallel composition proof techniques.

Theorem 4.7 (Decidability of $\equiv\_^{i\nu}\equiv$). *Given two processes P and Q and a relation $\mathcal{R}$, it is decidable whether $(P,Q) \in\ \equiv(\mathcal{R}^{i\nu})\equiv$.*

Proof. The proof given here is rather technical, still we do not give all details, in order to try to keep it readable. To decide $\equiv\_^{i\nu}\equiv$, we first look for a necessary condition for a pair (P,Q) of processes to belong to $\equiv(\mathcal{R}^{i\nu})\equiv$. Suppose then that the latter holds; this means that there exist σ, $\tilde{\tilde{x}}$, P_0, Q_0 s.t. $P_0\,\mathcal{R}\,Q_0$, $P = (\nu\tilde{\tilde{x}})(P_0\sigma)$ and $Q = (\nu\tilde{\tilde{x}})(Q_0\sigma)$.

Let us focus on P and P_0, and reason about the structure of normals forms. We have $(\nu\tilde{\tilde{x}})(\,(\nu x\tilde{\tilde{}}_{P_o}\langle P_0\rangle)\sigma\,) = (\nu\tilde{\tilde{x}}_P)\langle P\rangle$, which is equivalent to the existence of a substitution σ', injective on $\mathrm{fn}(\langle P_0\rangle)$, and a set of names $\tilde{\tilde{u}}$ disjoint from $\tilde{\tilde{x}}_{P_0}$ s.t.

$$\sigma'(\tilde{\tilde{x}}_{P_0} \uplus \tilde{\tilde{u}}) = \tilde{\tilde{x}}_P \quad\text{and}\quad \langle P_0\rangle\sigma' = \langle P\rangle\,. \tag{4.1}$$

Similarly, we can exhibit some σ'' and $\tilde{\tilde{v}}$ s.t.

$$\sigma'(\tilde{\tilde{x}}_{Q_0} \uplus \tilde{\tilde{v}}) = \tilde{\tilde{x}}_Q \quad\text{and}\quad \langle Q_0\rangle\sigma' = \langle Q\rangle\,. \tag{4.2}$$

Now getting back to σ, which is applied to both P_0 and Q_0, we further have that σ' and σ'' must coincide on the visible names, i.e.

$$\sigma' = \sigma'' \quad\text{on}\quad \mathrm{fn}(\langle P_0\rangle) \cap \mathrm{fn}(\langle Q_0\rangle) \setminus (\tilde{\tilde{x}}_{P_0} \uplus \tilde{\tilde{x}}_{Q_0} \uplus \tilde{\tilde{u}} \uplus \tilde{\tilde{v}})\,. \tag{4.3}$$

These are sufficient conditions, i.e. it holds that when we can exhibit σ', σ'', $\tilde{\tilde{u}}$ and $\tilde{\tilde{v}}$ s.t. 4.1, 4.2 and 4.3 are true, $P \equiv (\mathcal{R}^{i\nu}) \equiv Q$ (to show this, redo the above reasoning backwards). We can thus decide $\equiv\_^{i\nu}\equiv$: given P, Q, P_0 and Q_0, use AC matching to generate all possible (σ', σ''), and for each such pair, check the conditions above (which are obviously decidable). Iterate this process on all pairs of $\mathcal{R}$ until a successful (σ', σ'') is found; when no such pair can be found, fail. □

4.3 Adding up to Parallel Composition

We present here our most powerful proof technique for reasoning with π-calculus processes, which consists of the combination of four functions over relations.

Definition 16 ($\_^{i\nu|}$ function). The up to injective substitutions, restrictions, and parallel composition proof technique, written $\_^{i\nu|}$, is defined by the following function:

$$\mathcal{R}^{i\nu|} \stackrel{def}{=} \{(\,(\nu\tilde{\tilde{x}})\,((P\sigma)|T),\,(\nu\tilde{\tilde{x}})\,((Q\sigma)|T)\,).\; P\,\mathcal{R}\,Q \text{ and } \sigma \text{ is injective on } \mathrm{fn}(P|Q)\}\,.$$

Proposition 4.8 [San95]. *$\_^{i\nu|}$ is a respectful function.*

Proof. See the comments for Proposition 4.6.

Theorem 4.9 (Decidability of $\equiv\_^{i\nu|}\equiv$). *Given two processes P and Q and a relation $\mathcal{R}$, it is decidable whether $(P,Q) \in\ \equiv(\mathcal{R}^{i\nu|})\equiv$.*

Proof (Sketch). The up to restriction and up to injective substitutions proof techniques can smoothly be handled together in dealing with normal processes because they both involve reasoning only about the topmost restrictions (bodies of processes being just used to infer a matching substitution). Adding up to parallel composition somehow complicates the task. Without giving the full details of this proof, let us try to provide an intuition of the characterisation of $\equiv \_^{i\nu !} \equiv$ we establish, along the lines of the proof of Theorem 4.7.

The problem we are facing is, given two processes P and Q and a relation $\mathcal{R}$, to find a way to write

$$P \equiv (\nu \tilde{\tilde{u}})\,(P_0\sigma|T) \quad\text{and}\quad Q \equiv (\nu \tilde{\tilde{u}})\,(Q_0\sigma|T)$$

such that $(P_0, Q_0) \in \mathcal{R}$ for some processes P_0, Q_0, T, set of names $\tilde{\tilde{u}}$ and substitution σ. To achieve this, we use an extended AC matching algorithm, in order to isolate the common subcomponent T to be erased in both processes we are examining (note that, using the normalisation procedure, we can suppose that T is of the form $\langle T\rangle$). We then adapt criteria 4.1, 4.2 and 4.3 to decompose the topmost restrictions into those that are erased using up to restriction and those that remain, with the extra condition that no free name of T may be bound by the remaining restrictions. Indeed, we want to be able to write $P \equiv (\nu \tilde{\tilde{x}}_P)\,\langle P\rangle \equiv \nu\tilde{\tilde{u}}\,((\nu\tilde{\tilde{x}}_{P_0})\,\langle P_0\rangle \mid T)$ for some $\tilde{\tilde{u}}$ and T (in order to get rid of restrictions $\tilde{\tilde{u}}$ and of parallel component T), so we have to make sure that $\tilde{\tilde{x}}_{P_0} \cap \mathrm{fn}(T) = \emptyset$.

We will not enter further the details of this proof, whose principle is very similar to the proof of Theorem 4.7, while involving even heavier technical manipulations (due in particular to the extra side condition on free names mentioned above). □

The operational semantics of the π-calculus entails that committing replicated terms may give rise to infinite branches in the corresponding state space. As will be exemplified below, up to parallel composition may in some cases make it possible to "cut" these branches and close the bisimulation proof. Computational completeness (Theorem 2.3) is thus crucial in this situation.

5 Examples from a Prototype Implementation

In this Section, we experiment with a prototype implementation of our up-to bisimulation checking method for the π-calculus. We treat two non trivial examples which are in many ways paradigmatic for specification in the π-calculus, while being rather small (in no way do we claim them to be real-size case studies).

The methods presented so far have been implemented in a tool, written in O'Caml [Ler00], and running under Unix. The system allows the user to define a pair of processes and to check whether they are bisimilar by choosing between three up-to proof techniques and between strong bisimulation and *expansion* (see Section 2; note that the functions we have examined in Section 4 are also sound in the case of expansion). Facilities for interactively simulating the behaviour of a process are also provided. The tool is available at

`http://www.ens-lyon.fr/~hirschko/pi/.`

The implementation merely follows the algorithms presented above, with no particular effort towards efficiency. The breadth-first version of Mounier's algorithm, as described in in Figure 2.2, constitutes the kernel of the program. In a separate module, we have implemented the Term Rewrite System of Definition 13 following quite closely the definition. The difficult part in doing this is the application of rule R8, where we have to isolate two occurrences of the same term modulo α-conversion (names are implemented using identifiers, which facilitates interaction with the user, but leads to an explicit treatment of α-conversion) and AC properties of parallel composition. We therefore had to define a function implementing a form of AC matching taking into account possible renamings of bound variables. Implementation of this function represents probably the most tricky part in the construction of our tool. This machinery is also necessary to decide whether a pair of processes belongs to the closure of a relation up to a given proof technique (properties 4.5, 4.7 and 4.9).

As said above, we have made no particular effort as far as the efficiency of our implementation is concerned. While we may hope that the up-to techniques help in controlling the phenomenon of state explosion when exploring the state space, we believe that the AC unification procedure mentioned above inevitably leads to a dramatic explosion both in time and space. AC matching is indeed known to be a difficult problem; we are not aware of any theoretical study regarding the complexity of the problem we are facing here, where AC properties of operator $|$ are handled together with the phenomenon of binding in the π-calculus. Due to their small size, the case studies we have been working on (see below) do not provide any relevant information on this issue. It is however likely that, to be applied to real world examples, the methods presented in this work deserve a specific study towards performance enhancement.

5.1 Persistent Data Structures and Sharing

For our first example, we draw inspiration from the treatment of lists exposed in [Mil91] to illustrate some considerations about the representation of persistent data structures in the π-calculus. We consider a very simple system, consisting in a cell giving access to a boolean value. This will however be sufficient to reason about *sharing properties* of resources in the π-calculus, and, in doing this, to perform non-trivial bisimulation proofs.

Along the lines of the classical construction in the λ-calculus, a boolean value in the π-calculus is a process

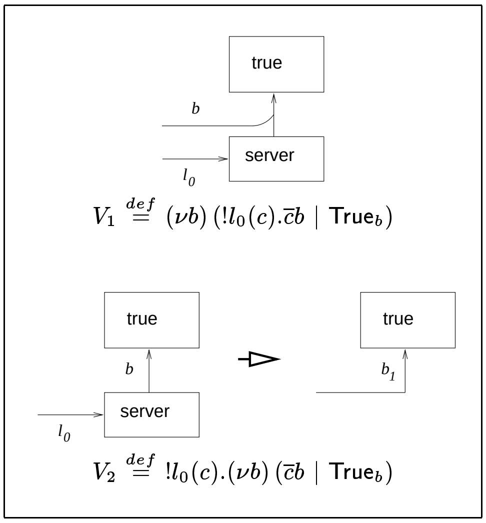

Fig. 5.1. Giving Access to a Boolean Value

located at some channel b, where it is liable to receive a pair of names and to emit back on one of these according to the truth value it represents. We hence have, for booleans located at b:

$$\mathsf{True}_b \stackrel{def}{=} !b(t,f).\overline{t} \quad \text{and} \quad \mathsf{False}_b \stackrel{def}{=} !b(t,f).\overline{f}$$

The *persistent* behaviour of these data is obtained by replicating the above terms. We can also define an *if-then-else* construct, which tests a boolean situated at b, as follows:

$$\underline{\text{if}}\ b\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q \stackrel{def}{=} (\nu t,f)(\overline{b}[t,f] \mid t.P \mid f.Q)\,.$$

(Note that to avoid possible interferences with other processes interacting with b, we send two *private* names t and f).

We now want to define an *interface* to a given boolean value, controlling access to it. According to [Mil91], one can think of two different protocols to do so, depicted on Figure 5.1, which also provides the corresponding π-calculus definitions. By reception of a request on l_0, V_1 sends back on c the address of the boolean b, while V_2 generates a *fresh copy* of the boolean cell, and sends the new address b_1 (generated by the (νb) construct) for the private usage of the requester.

We make a few comments on the processes we have defined. Although being very elementary, the definitions of V_1 and V_2 are built up using mechanisms that are paradigmatic in writing π-calculus processes, in particular w.r.t. name passing. The subterm True_b (see above) is of the form $!a(\widetilde{x}).P$: such terms are usually called *resource processes* in the π-calculus ($!a(\widetilde{x}).P$ represents the resource P, depending on parameters $\widetilde{x}$, located at a). Resource processes are ubiquitous in the study of π-calculus: they can be found in particular in the encodings of the λ-calculus (see e.g. [San94]), where application is represented as (the translation of) the function having access to its argument as a resource. Such terms also arise in the study of higher-order π-calculus, of object calculi, and of data structures (like in the present study, drawn from [Mil91]).

Moreover, the way terms V_1 and V_2 are constructed "around" the boolean value, is directly inspired from the implementation of the *Cons* constructor for lists in [Mil91]. The associated behaviour is reminiscent from an object-oriented programming sty-le, where we give access to a set of processes (values or methods) through an interface, whose rôle is to establish a kind of protocol for interacting with these processes.

Let us go back to our example. As stated in [Mil91], the difference in the protocols used by V_1 and V_2 is reflected on *sharing properties* of the data structures. The problem of value sharing can be expressed in terms of security: suppose we have an agent A, willing to test any number of times the resource V (implemented as V_1 or V_2) located at l_0, the outcome of the test being a signal on either of two channels u or v. This agent can be represented by the process

$$A \stackrel{def}{=} !\overline{l_0}c_1.c_1(b).\underline{\text{if}}\ b\ \underline{\text{then}}\ \overline{u}\ \underline{\text{else}}\ \overline{v}\,,$$

where c_1 is a name that is specific to A for its communications with V. Suppose now that an evil agent E is willing to interfer with the communications of A, by interrogating V and trying to send the value False as if this was the value of the cell located at l_0. To do this, E needs to interrogate once V, to acquire the knowledge of name b where the value is located, and then "mimick" False along b. Hence:

$$E \stackrel{def}{=} \overline{l_0}c_2.c_2(b).\mathsf{False}_b\,.$$

The whole system we consider is made of the parallel composition of the resource (being either V_1 or V_2), "innocent" agent A, and "evil" agent E:

$$\mathrm{Sys}_i \stackrel{def}{=} (\nu l_0, c_1, c_2)\,(\,E \mid V_i \mid A\,) \qquad i = 1,2\,.$$

We verify that, depending on which representation we take for the persistent cell V, E can either disturb the communications performed by A (which corresponds to a "security flaw"), or be innocuous. To this end, we compare the behaviour of the term Sys_i for $i = 1,2$ with the behaviour of the system where agents E and A have their own private copy of the resource (and hence cannot interfer), given by:

$$\mathrm{Sys}'_i \stackrel{def}{=} (\nu l_0, c_2)\,(E \mid V_i) \mid (\nu l_0, c_1)\,(A \mid V_i) \quad i = 1,2\,.$$

Figure 5.2 presents the tool sessions corresponding to both experiments (the syntax for processes is rather natural: `^` stands for abstraction, `[]` for output and `()` for input). Interaction with the tool is as follows: the user enters two processes (keywords `Left` and `Right`), and starts the checking procedure. A few commands make it possible to switch between strong and weak bisimulation, and between the various up-to techniques presented above (which corresponds to a change of the plug-in in the "boxed" parts of Figure 2.2). When an up-to bisimulation relation can be found, it is displayed. The results we obtain in Figure 5.2 can be analysed as follows:

Case $\mathbf{Sys}_1$. In this case, a location b for the value True is created once for all and transmitted to every process that interrogates the value at l_0. This way, even in the case where A performs only one request at l_0, agent E can send the wrong value False to A, and Sys_1 and Sys_1' are not bisimilar.

Case $\mathbf{Sys}_2$. In this case, a new name b is created each time the list is interrogated at l_0, and hence processes A and E cannot interfer. Even in the case where an infinite number of processes interrogate the cell, we can prove that Sys_2 and Sys_2' are bisimilar.

When the tool performs this verification, it exploits the up to parallel composition proof technique to erase copies of the resource process each time they appear in both terms: an infinite proof is thus replaced by a finite one. It should be stressed that other existing systems for the π-calculus cannot handle this proof, i.e. they enter an infinite loop.

To conclude on this example, let us remark that "nocive" behaviours like the one E exhibits can be ruled out using type systems for the π-calculus. For example, [San97] introduces a name discipline where a received name cannot be used in input, thus preventing E to be well-typed.

5.2 The Alternating Bit Protocol

We now focus on another example: the well-known *alternating bit protocol* (ABP), which is probably the most widely used benchmark for verification systems in the field of concurrency. Our purpose here is not to study it *per se* (the ABP can be verified using e.g. CCS [Mil89]), but instead to use it to shed light on issues related to the question of encodings in the π-calculus, as will be demonstrated below. Indeed, in treating this example, the up-to techniques help us in reasoning about the ABP within the relatively small language we have adopted (no data structures, no choice operator).

To study the ABP, we shall use expansion (cf Section 2): $\lesssim$ is more realistic than (strong) bisimulation for reasoning about larger case-studies such as protocols. It allows one to prove that a given term "respects" a behaviour, modulo some extra τ steps (hidden in $\stackrel{\mu}{\Rightarrow}$ transitions).

We first informally introduce the protocol, then present an implementation in the π-calculus adapted from [Mam99], and discuss the encodings we use in order to automatically perform the correctness proof. In discussing encodings, we want to stress the fact that the question of defining encodings in the framework of verification departs from the traditional issue of studying the *expressiveness* of the formalism (which is what en-

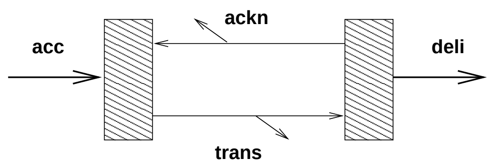

Fig. 5.3. The Alternating Bit Protocol

codings are often used for), due to the central question of avoiding divergence in the execution of processes.

The Protocol Figure 5.3 presents the entities involved in the communication protocol: a first agent receives a message (whose content is not taken into account here) on channel acc. It then transmits this message on a channel called trans to a second agent. But channel trans is unreliable, and some messages transiting on it may get lost: to handle that situation, a boolean tag is attached to the message being sent on this channel, and the first agent keeps sending this information on trans. By reception, the second agent transmits the message on deli and is willing to send back the received boolean as an acknowledgment to the first agent (on ackn). Channel ackn being lossy as well, the second agent has to do this repeatedly. When the acknowledgment eventually arrives, a new cycle can start with the negated value of the boolean, so that the first agent can tell when the message has actually been received.

Booleans We have already seen how booleans are implemented in the π-calculus; we need here some extra machinery to define the protocol, consisting in an operator to test for equality of two booleans and a negation construct. The former is easily defined as follows:

$$[\![b = b']\!]_{b_0} \stackrel{def}{=} b_0(T,F).\,(\nu t,f,t',f')(\overline{b}[t;f] \mid \overline{b'}[t';f'] \mid t.(t'.\overline{T}|f'.\overline{F}) \mid f.(t'.\overline{F}|f'.\overline{T}))$$

(double square braces $[\![\,]\!]$ denote the encoding function). It has to be noted that once the test has been performed, non-chosen branches are automatically erased by the normalisation procedure, using rule $\mathsf{R7'}$.

Surprisingly enough, encoding the negation construct is more problematic. The obvious encoding would be:

$$[\![\neg b]\!]_{b'} \stackrel{def}{=} !b'(t,f).\overline{b}(f,t)\,.$$

However, it turns out that each time the protocol is restarted with a negated boolean value, a transmitting process of the form above is added to the term we are examining. In other words, we have no way to garbage collect two consecutive negations, by saying that $\neg\neg b$ reduces to b. Therefore, adopting the above encoding would lead to an infinite growth of the relation when checking the protocol. To avoid this, we use a trick, and parametrise the whole protocol over *two* (opposite) boolean values, a call with a negated boolean being achieved by exchanging the two parameters.

```
Please enter your input:
> Left (^c1)(^l0)(((^b)(!l0(c).c[b]|!b(t;f).t[])) | (^c2)l0[c2].c2(b).!b(t;f).f[]
 | !l0[c1].c1(b).(^t)(^f)(b[t;f]|t.u[]|f.v[]))

Please enter your input:
> Right (^c1)(^l0)(((^b)(!l0(c).c[b]|!b(t;f).t[])) | (^c2)l0[c2].c2(b).!b(t;f).f[])
 | (^l0)(((^b)(!l0(c).c[b]|!b(t;f).t[])) | !l0[c1].c1(b).(^t)(^f)(b[t;f]|t.u[]|f.v[]))

Please enter your input:
> Check
The processes are not bisimilar.
```

```
Please enter your input:
> Left (^l0)(((!l0(c).(^b)c[b].b(t;f).t[])) | l0[c2].c2(b).b(t;f).f[]
 | !l0[c1].c1(b).(^t)(^f)(b[t;f]|t.u[]|f.v[]))

Please enter your input:
> Right (^l0)(((!l0(c).(^b)c[b].b(t;f).t[])) | l0[c2].c2(b).b(t;f).f[])
 | (^l0)(((!l0(c).(^b)c[b].b(t;f).t[])) | !l0[c1].c1(b).(^t)(^f)(b[t;f]|t.u[]|f.v[]))

Please enter your input:
> Check
The processes are bisimilar.
Size of the relation is 1:
...
```

Fig. 5.2. Verifying Systems Sys_1 and Sys_2

$$
\begin{aligned}
\mathit{Send} &\stackrel{def}{=} !\mathsf{send}(b_1,b_2).\mathsf{acc}.\overline{\mathsf{trans}}[b_2;b_1].\overline{\mathsf{wait}}[b_2;b_1] \\
\mathit{Wait} &\stackrel{def}{=} !\mathsf{wait}(b_1,b_2).(\mathsf{ackn}(b_1',b_2').\overline{\mathsf{send}}[b_1';b_2'] \\
&\qquad + \mathsf{loss}.\overline{\mathsf{trans}}[b_1;b_2].\overline{\mathsf{wait}}[b_1;b_2]) \\
\mathit{Receive} &\stackrel{def}{=} !\mathsf{rec}(b_1,b_2).\mathsf{trans}(b_1',b_2').\underline{\text{if}}\ b_1 = b_1' \\
&\qquad \underline{\text{then}}\ \ \overline{\mathsf{ackn}}[b_1;b_2] \mid \overline{\mathsf{rec}}[b_1;b_2] \\
&\qquad \underline{\text{else}}\ \ \overline{\mathsf{deli}}.(\overline{\mathsf{ackn}}[b_2;b_1] \mid \overline{\mathsf{rec}}[b_2;b_1]) \\
\mathit{Noise} &\stackrel{def}{=} !\mathsf{noise}.(\,\mathsf{trans}(b,b').\overline{\mathsf{loss}}.\overline{\mathsf{noise}} \\
&\qquad +\ \mathsf{ackn}(b,b').\overline{\mathsf{loss}}.\overline{\mathsf{noise}}\,) \\
\mathit{System} &\stackrel{def}{=} (\nu\,\mathsf{trans},\mathsf{ackn},\mathsf{send},\mathsf{wait},\mathsf{rec},\mathsf{loss},\mathsf{noise},b,c) \\
&\quad (\,\mathit{Send} \mid \mathit{Wait} \mid \mathit{Receive} \mid \mathit{Noise} \mid \mathsf{True}_b \mid \mathsf{False}_c \\
&\qquad \mid \overline{\mathsf{send}}[b,c] \mid \overline{\mathsf{rec}}[b,c] \mid \overline{\mathsf{noise}}\,) \\
\mathit{Specif} &\stackrel{def}{=} (\nu s)\,(!s.\mathsf{acc}.\overline{\mathsf{deli}}.\overline{s} \mid \overline{s})
\end{aligned}
$$

Fig. 5.4. Modelling the Alternating Bit Protocol

Implementation of the Protocol Figure 5.4 presents our implementation of the protocol in the π-calculus[2]. The first agent's behaviour is given by processes *Send* and *Wait*, while the second agent is described by process *Receive*. Message losses on trans and ackn are represented by process *Noise*, that sends a signal on a channel loss whenever a message gets lost. Finally, process *Specif* is an ideal specification of the protocol's behaviour: it receives a signal on acc, sends one on deli, and starts again by sending a signal on the trigger channel s (this mechanism for implementing recursive behaviours is extensively used in our encoding; it is defined in [Mil91]).

Choice We are left with the representation of the choice construct, written $+$, which is used in Figure 5.4: a process of the form $P+Q$ is liable to behave like P or Q, in an exclusive way. Encodings of this operator in a choice-free π-calculus have been studied in [NP96,Nes97]; the general idea is to represent each branch of the choice by a parallel component, and to implement a lock mechanism to prevent other branches to fire when one branch has committed. The solutions of [NP96,Nes97] do not fit for our purposes, though, due to the treatment of non-chosen branches. As in the case of boolean negation, we want to be able to syntactically erase a dead branch in a choice construct *as soon as the choice has taken place*, to prevent the relation from growing. We have not been able to coin a method to achieve this using the general solutions of [NP96,Nes97], and therefore we restrict ourselves to simpler solutions.

A first possibility is given by *internal choice*, as defined in [NP96] (we only need here to represent choice between two input-prefixed terms):

$$[\![a(\widetilde{x}).P \,+\, b(\widetilde{y}).Q]\!] \stackrel{def}{=} (\nu l)\,(\bar{l} \mid l.a(\widetilde{x}).P \mid l.b(\widetilde{y}).Q)\,.$$

In this case, the choice is made *within* process $a(\widetilde{x}).P + b(\widetilde{y}).Q$, and the non-chosen branch is successively erased using rule R7'. This solution cannot however be adopted in our case, since for example process *Wait* of Figure 5.4 does not have the ability to decide whether it can receive an acknowledgment on ackn or if a loss has taken place.

We instead adopt a symmetrical method, that we call *external choice*, by taking advantage of a property of the terms we use to encode the ABP: it is always the case, in the run of the protocol, that when a choice occurs, at most one branch of the choice can be fired[3] – hence *externality*: the context "chooses" the branch. We can thus use the following encoding:

$$
\begin{aligned}
&[\![a(\widetilde{x}).P \,+\, b(\widetilde{y}).Q]\!] \stackrel{def}{=} \\
&\qquad (\nu l)\,(\bar{l} \mid a(\widetilde{x}).l.\bar{b}(\widetilde{x}).P \mid b(\widetilde{y}).l.\bar{a}(\widetilde{y}).Q)\,.
\end{aligned}
$$

Again, it has to be noted that this encoding of choice is correct only because of the aforementioned property of the surrounding context when a choice is ready to commit.

We have now completely defined the encoding of the protocol into our simple language. The corresponding term is fed to the tool like in the previous case study, as well as the definition of *Specif*. We then indicate that we want to check expansion up to structural equivalence and parallel composition, and we finally verify automatically that *Specif* $\lesssim$ *System*. The positive answer of the system comes with the expansion relation, which corresponds quite closely to the bisimulation relation from Milner's treatment in CCS [Mil89].

[2] The language used on Figure 5.4 could be seen as a variant of value-passing CCS, using replication instead of recursive definitions. However, the encodings we use exploit the full power of the π-calculus.

[3] Note that this property holds because of the signal on channel loss when a failure takes place; such synchronisations can indeed be interpreted as a modelisation of a timer mechanism to detect channel failures.

6 Conclusion

Discussion We have presented a methodology for the transfer of a collection of proof techniques for bisimulation from the status of theoricians' methods onto the context of verification, by designing appropriate algorithms, and providing an implementation of these to evaluate the outcome of such an effort. It appears that both in terms of efficiency (reduction of the size of the relations one has to handle) and of expressivenes (the ability to transform infinite proofs into finite ones, and in doing so, to treat cases that existing tools for the π-calculus cannot handle), the benefits one can expect from using the up-to techniques for bisimulation for verification are not to be neglected. Indeed, we have been able to perform on machine a few proofs concerning non-trivial π-calculus terms. This can be seen as a step towards the mechanisation of reasoning over systems involving complex concurrent behaviours.

It can however be questioned whether a fully automatic approach for verification for the π-calculus and related formalisms is realistic, due to the great expressiveness of such models. Indeed, it seems difficult to design rigourous methods for verification that would be general enough to treat a large subclass of terms (in particular, infinite behaviours should probably not left out, as is often the case). Typically, the problem of finding a non-trivial class of terms for which our checking method is *complete* seems difficult. As an alternative to this point of view, efforts have been conducted using (interactive) theorem proving [HMS99,Hir97]. More generally, in the field of verification of concurrent systems, it seems that the integration of model checking techniques with theorem proving has recently gained interest. In this perspective, a methodology like ours, which is based on the "on the fly" checking method, fits within the scenario of such a cooperation. Indeed, the objects we work with on the verification side are the same on the theorem proving side: terms (as opposed e.g. to BDDs encodings of state spaces). This departs from the approach consisting in using a logical framework to "justify" some crucial steps (typically, the construction of abstractions) in a traditional model checking method as is done e.g. in [AAB+99]. An advantage of the latter methodology, though, is that it allows one to benefit from the considerable experience in building efficient and robust verification tools.

Future Work In order to treat larger examples, we probably should enhance our tool. The language we work with could be enriched (e.g. constructors like the choice operator should belong to the core of the specification language, instead of being encoded into it). It is not clear though whether the up-to techniques for the π-calculus of Section 4 could easily be adapted to these extensions. Alternatively, one could also think of integrating our set of up-to techniques to an existing tool like the Mobility WorkBench [VM94].

Further improvements of our method can also be provided by developping the algebraic treatment of terms in the normalisation procedure of paragraph 4.1. One can in particular add laws to check *expansion up to expansion*, such as e.g.

$$(\nu b)\,(!a(\widetilde{x}).\overline{b}[\widetilde{x}] \mid !b(\widetilde{y}).\overline{c}[\widetilde{y}]) \;\gtrsim\; !a(\widetilde{x}).\overline{c}[\widetilde{x}]\,,$$

which is a basic property of so-called *link processes* (see e.g. [MS98]; such processes actually embed a form of aliasing on names). Similar rewrite rules would for example make it possible to effectively reduce $\neg\neg b$ into b in paragraph 5.1 above. Work is in progress to enrich this way the definition of $\equiv$. The hope in doing so is to be able to capture a significant amount of "standard" idioms that are used for programming in the π-calculus. Typically, one would expect to perform some elementary reasoning on π-calculus representations of data structures, along the lines of paragraph 5.1. There exist well understood encodings for data structures in the π-calculus [Mil91]. As such encodings generally obey a kind of *programming discipline* in the managment of names, we could reasonably think of building a "library" of bisimilarity results about this kind of terms. This would be interesting e.g. in relation with a theorem prover formalisation of non-trivial systems such as protocols. Indeed, as an outcome of [ER99], it turns out that in such a mechanisation, many proofs are made up from a "clever" part, where the very mechanism of the protocol is examined, and a more "technical" component, dealing with elementary properties of the data structures being used. One could thus think of adopting a verification approach for the latter, in order to limit (time-consuming) interactive reasoning only on the protocol, without loosing too much confidence about the final proof. This kind of interaction between an automatic mode and human guidance could also be used to provide proof hints about the application of certain up-to techniques (typically, up to expansion and up to bisimilarity) at some points along bisimulation proofs.

The rather informal statements we have made above regarding programming discipline and the way names are used in the π-calculus could be rigorously treated using *type systems* [San97,PS]. Understanding how such systems can be integrated to the verification method seems a promising direction in which the present work could be extended.

Finally, let us stress that we have tried to "factorise" in this presentation the treatment of up-to techniques, in order to keep it independent from their application to the π-calculus. Our hope in doing so is to serve the use of up-to bisimulation verification for other process algebras.

Acknowledgments We benefited from discussions with M. Boreale, D. Sangiorgi, and B. Sudret. Accurate reading by the anonymous referees helped a lot to improve the paper.

References

[AAB+99] P. A. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, and P. Habermehl Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability. In *Proc. of CAV '99*, LNCS, 1999.

[AG99] M. Abadi and A. Gordon. A calculus for cryptographic protocols: the spi calculus. *Information and Computation*, 148(1):1–70, 1999.

[Ama97] R. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. of Coordination '97*, volume 1282 of *SLNCS*, 1997.

[BDNP99] M. Boreale, R. De Nicola, and R. Pugliese. Proof Techniques for Cryptographic Processes. In *Proc. of LICS '99*. IEEE Computer Society Press, 1999.

[Bor98] M. Boreale. On the Expressiveness of Internal Mobility in Name-Passing Calculi. *TCS*, 195(2):205–226, 1998.

[Bry94] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of of 22nd Design Automation Conference*, pages 688–694. IEEE Computer Society Press, 1994.

[BS98] M. Boreale and D. Sangiorgi. Bisimulation in name-passing calculi without matching. In *Proc. of LICS '98*. IEEE Computer Society Press, 1998.

[Car99] L. Cardelli. Abstractions for Mobile Computations. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *LNCS*. Springer Verlag, 1999.

[Cau90] D. Caucal. Graphes canoniques de graphes algébriques. *Informatique Théorique et Applications (RAIRO)*, 24(4):339–352, 1990.

[CG98] L. Cardelli and A. Gordon. Mobile Ambients. In *Proceedings of FOSSACS '98*, LNCS, 1998.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Proceedings of CAV'96*, volume 1102 of *LNCS*, pages 394–397. Springer Verlag, 1996.

[ER99] J. Esparza and C. Röckl. Proof-checking protocols using bisimulations. In *Proc. of CONCUR '99*, volume 1664 of *LNCS*, 1999.

[FG96] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of POPL '96*. ACM Press, 1996.

[FGM+97] G. Ferrari, G. Ferro. S. Gnesi, U. Montanari, G. Ristori, and M. Pistore. An automata based verification environment for mobile processes. In *Proceedings of TACAS '97*, LNCS. Springer Verlag, 1997.

[FM91] J.-C. Fernandez and L. Mounier. "On the fly" verification of behavioural equivalences and preorders. In *Proceedings of CAV'91*, LNCS, 1991.

[FM97] C. Fournet and L. Maranget. *The Join-Calculus language - documentation and user's manual.* INRIA, Rocquencourt, 1997.

[Fou98] C. Fournet. *Le join-calcul : un calcul pour la programmation rpartie et mobile.* PhD thesis, INRIA, 1998.

[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.

[Hir98] D. Hirschkoff. Automatically Proving Up To Bisimulation. In *Proceedings of MFCS'98 Workshop on Concurrency*, volume 18 of *ENTCS*, 1998.

[Hir99a] D. Hirschkoff. *Mise en œuvre de preuves de bisimulation.* PhD thesis, ENPC, Champs sur Marne, France, January 1999. in french.

[Hir99b] D. Hirschkoff. On the Benefits of Using the Up To Techniques for Bisimulation Verification. In *Proc. of TACAS '99*, volume 1579 of *LNCS*, 1999.

[HMS99] F. Honsell, M. Miculan, and I. Scagnetto. Pi-calculus in (Co)Inductive Type Theories. *TCS*, 1999. to appear.

[KS90] P.C. Kanellakis and S.A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[Ler00] X. Leroy. *The Objective Caml Documentation.* INRIA Rocquencourt, 2000.

[Mam99] B. Mammas. *Méthodes et outils pour les preuves compositionnelles de systèmes parallèles.* PhD thesis, Univ. Paris 6, 1999. (in french).

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

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

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

[MS92] R. Milner and D. Sangiorgi. Techniques of "weak bisimulation up to". In *CONCUR '92*, volume 630 of *LNCS*, 1992.

[MS98] M. Merro and D. Sangiorgi. On asynchrony in name-passing calculi. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*. Springer Verlag, 1998.

[Nes97] U. Nestmann. What is a 'good' encoding of guarded choice? In *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*, 1997.

[New42] M. H. A. Newmann. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[NP96] U. Nestmann and B. C. Pierce. Decoding choice encodings. In *Proceedings of CONCUR '96*, volume 1119 of *LNCS*. Springer Verlag, 1996.

[Par81] D. Park. Concurrency and automata on infinite sequences. *LNCS*, 154:561–572, 1981.

[PS] B. Pierce and D. Sangiorgi. Behavioural equivalence in the polymorphic pi-calculus. *Journal of the ACM*. to appear.

[PS81] G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of ACM*, 2(28):233–264, April 1981.

[PS96] M. Pistore and D. Sangiorgi. A partition refinement algorithm for the π-calculus. In *Proceedings of CAV '96*, volume 1102 of *LNCS*, 1996.

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

[PT98] B. Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[RS99] C. Röckl and D. Sangiorgi. A π-calculus process semantics of concurrent idealised algol. In *Proc. of FOSSACS '99*, volume 1578 of *LNCS*, 1999.

[San94] D. Sangiorgi. The Lazy Lambda Calculus in a Concurrency Scenario. *Information and Computation*, 111(1), 1994.

[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.

[San97] D. Sangiorgi. The name discipline of uniform receptiveness. In *Proc. of ICALP '97*, volume 1256 of *LNCS*, pages 303–313. Springer Verlag, 1997.

[Vas94] V. Vasconcelos. Typed concurrent objects. In *Proc. of ECOOP '94*, volume 821 of *LNCS*, 1994.

[VM94] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In *Proc. of CAV'94*, volume 818 of *LNCS*, pages 428–440, 1994.