On the Benefits of Using the Up–To Techniques for Bisimulation Verification

Daniel Hirschkoff

Novembre 1998

N^o 98-138

On the Benefits of Using the Up–To Techniques for Bisimulation Verification

Daniel Hirschkoff

Résumé

Nous présentons un outil qui permet la vérification de propriétés de bisimulation utilisant les techniques de preuve up–to (bisimulation à congruence structurelle près, aux restrictions près, à composition parallèle près). Parmi ces techniques, la possibilité de travailler "à composition parallèle près" est particulièrement intéressante dans la mesure où elle permet de raisonner sur des termes dont l'espace d'états est infini. Afin de l'exploiter au mieux, nous adaptons l'algorithme de vérification à la volée, *de manière à garantir à nos méthodes une forme de complétude computationnelle. Les techniques de preuve up–to s'avèrent utiles pour exploiter la puissance expressive du π–calcul, comme le montrent deux études de cas non triviales, concernant la représentation des structures de données persistantes dans le π–calcul d'une part, et la vérification du protocole du bit alterné d'autre part. Ce dernier exemple nous permet de revenir sur des encodages connus en π–calcul, à la lumière des critères propres au contexte de la vérification automatique.*

Abstract

We advocate the use of the up–to techniques for bisimulation in the field of automatic verification. To this end, we develop a tool to perform proofs using the up to structural congruence, the up to restrictions and the up to parallel composition proof techniques for bisimulation between π–calculus terms. The latter technique is of particular interest because it allows the user to reason on infinite state space processes. To use it in full effect, we adapt the "on the fly" bisimulation checking algorithm, yielding to a form of computational completeness. The usefulness of these techniques in dealing with the expressive power of the π–calculus is illustrated on two non trivial examples, namely the treatment of persistent data structures and the alternating bit protocol. These examples are also good opportunities to study how well–known π–calculus encodings behave in the framework of automatic verification.

Introduction

This paper studies the applicability of the so–called up–to techniques for bisimulation in the field of verification. Bisimilarity, and its correlated proof method bisimulation, have become popular notions of equivalence used to reason about concurrent systems. While having a simple and clear mathematical definition, they are far from being straightforward to handle in the framework of automatic verification, due to their richness and to the many subtle phenomena they are able to catch in the study of concurrency. The up–to techniques for bisimulation are presented in [San95]; they can simplify bisimulation proofs by reducing the size of the relations one has to consider. More precisely, an up–to technique is represented by a function $\mathcal{F}$ from relations to relations, such that proving that processes related in a relation $\mathcal{R}$ evolve to processes related in $\mathcal{F}(\mathcal{R})$ is enough to show that $\mathcal{R}$ is contained in bisimilarity. One can thus consider relations that are smaller than bisimulations, the gap being filled by the application of $\mathcal{F}$.

In this paper, we concentrate on bisimulation between π–calculus terms; the π–calculus [Mil91] has become a widely accepted algebra for modelling concurrency, and has demonstrated a great expressive power. It is our belief that, to use π–calculus as a specification language in the field of verification, one should be able to cope with the richness of this formalism, in particular by designing *specific* verification techniques for it, and not only transpose methods that are applied to less expressive formalisms. The Mobility WorkBench [VM94] is a good example of a verification tool developped specifically for the π–calculus (and more recently for its evolution *fusion calculus*), taking advantage in particular of the nice characterisations given by *open bisimulation* [San96]. With these ideas in mind, the up–to methods seem to be good candidates for this task: the bisimulation up to bisimilarity proof technique, and more importantly the up to restrictions and the up to parallel composition proof techniques, have been developped in the theoretical study of π–calculus, and typically allow one to deal with name extrusion (up to restrictions proof technique) and with replication (up to parallel composition proof technique).

Feasability of the automation of the up–to techniques has been adressed in [Hir98], where some methods to decide if a pair of processes belongs to a relation up to structural congruence, up to restrictions, and up to parallel composition, are introduced. These methods lead to the definition of a general verification algorithm for up–to bisimulation between π–calculus terms, embedded in a prototype tool. In the present work, we extend the capabilities of the system, and, more importantly, try to evaluate the outcome of such an effort through the study of two non trivial examples that exploit the π–calculus' expressive power. With respect to [Hir98], the current version of the system is provided with a richer notion of structural congruence, and, moreover, the general bisimulation verification algorithm is modified in order to take into account the way the up to parallel composition proof technique works.

The plan of the paper is as follows: in the Section 1, we introduce the formal background on π–calculus and the up–to techniques for bisimulation. Section 2 is devoted to the description of our implementation, and of the general bisimulation algorithm we use to handle the up to parallel composition proof technique in a computationally complete fashion. We then describe two case studies: persistent data structures, as introduced by Milner [Mil91], are examined in Section 3, while Section 4 is devoted to the study of the alternating bit protocol, in an encoding adapted from [Mam98]. We finally conclude by discussing future work.

1 Preliminaries

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. We then define bisimulation, together with the up–to techniques for bisimulation.

Definition 1 (Syntax) *Given an infinite countable set of* names $\mathcal{N}$*, ranged over by* $a, b, c, \ldots, p, q, \ldots, x, y, \ldots$*, we range over (possibly empty) name lists with* $\vec{a}, \vec{b}, \ldots$*, and we define* prefixes*, ranged over by* α, β*, as follows:*

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

Processes*, ranged over with* $P, Q, R, \ldots$*, are then defined as follows:*

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

Prefixed processes are either receptions (prefix $a(\vec{b})$) or emissions ($\overline{a}[\vec{b}]$); $\mathbf{0}$ is the inactive process; the restriction operator ν makes a name private to the restricted process; parallel composition is written $|$, and ! stands for replication, allowed only on prefixed processes; intuitively, a replicated process represents any number of copies of this process put in parallel.

Conventions and notations: In prefixes $a(\vec{b})$ and $\overline{a}[\vec{b}]$, a and $\vec{b}$ 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 $\mathbf{0}$. Free and bound names are defined by saying that restriction and abstraction (embodied in the input prefix) are binding operators. As usual, we shall silently use α–conversion to avoid capture of bound names.

Definition 2 (Transition system) *The operational semantics of π–calculus terms is defined as a labeled transition system.* Actions*, ranged over with μ, μ', are given by the following syntax:*

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

which reads as follows: an action is either a reception*, a (possibly bound) output, or the silent action τ, denoting internal communication. Bound and free names of actions are defined as usual.*

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 1 (symmetrical versions of rules PAR_l *and* CLOSE_1 *are omitted; :: is the constructor used to add an element to a list). Note that we adopt an "early" version of the operational semantics.*

$$(\text{INP})\quad a(\vec{b}).P \xrightarrow{a(\vec{c})} P_{\vec{b}:=\vec{c}} \qquad\qquad (\text{OUT})\quad \overline{a}[\vec{b}].P \xrightarrow{\overline{a}[\vec{b}]} P$$

$$(\text{BANG})\quad \frac{\alpha.P \xrightarrow{\mu} P'}{!\alpha.P \xrightarrow{\mu} !\alpha.P|P'} \qquad (\text{RES})\quad \frac{P \xrightarrow{\mu} P'}{(\nu x)\,P \xrightarrow{\mu} (\nu x)\,P'}\ x \notin n(\mu)$$

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

$$(\text{OPEN})\quad \frac{P \xrightarrow{(\nu\vec{b'})\,\overline{a}[\vec{b}]} P'}{(\nu t)\,P \xrightarrow{(\nu t::\vec{b'})\,\overline{a}[\vec{b}]} P'}\ t \neq a, t \in \vec{b} \setminus \vec{b'}$$

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

Figure 1: Operational semantics

The notion of semantical equivalence on processes we are interested in is bisimilarity, defined as follows:

Definition 3 (Bisimulation, bisimilarity) *A relation $\mathcal{R}$ over processes is a* bisimulation *iff, whenever $P \mathcal{R} Q$ and $P \xrightarrow{\mu} P'$, there exists a process Q' s.t. $Q \xrightarrow{\mu} Q'$ and $P' \mathcal{R} Q'$, and conversely for the transitions of Q. Bisimilarity, written $\sim$, is the greatest bisimulation.*

Definition 4 (Structural congruence) Structural congruence*, written $\equiv$, is the smallest equivalence relation on π–calculus terms that is a congruence generated by the following rules:*

$$P|\mathbf{0} \equiv \mathbf{P} \qquad P|Q \equiv Q|P \qquad P|(Q|R) \equiv (P|Q)|R \qquad (\nu x)\,\mathbf{0} \equiv \mathbf{0}$$
$$!\alpha.P|\alpha.P \equiv !\alpha.P \quad !\alpha.P|!\alpha.P \equiv !\alpha.P \quad (\nu x)\,P \mid Q \equiv (\nu x)\,(P|Q) \;\; \textit{if}\; x \notin fn(Q)\,.$$

Note that rule for processes of the form $!\alpha.P|!\alpha.P$ is new with respect to the usual definition of structural congruence, and define a straightforward extension of this relation. Note as well that $\equiv$ is obviously included in $\sim$.

We shall actually use a slightly enriched version of structural congruence, where we add the following law:

$$(x \text{ occurs in } \alpha \text{ in subject position}) \;\Rightarrow\; (\nu x)\,\alpha.P \;\equiv\; \mathbf{0}\,.$$

This rule is usually not included in the definition of structural congruence, because its signification is too "semantical", i.e. it expresses a behavioural property of a process rather than a geometrical one, as structural congruence is intended to do. However, in the framework of the up to bisimilarity proof technique, it is preferable to equate as many terms as possible, hence an extended version of structural congruence is of interest; moreover, this particular law will turn out to be very useful in the example treated in Section 4. We extend the notation $\equiv$ to our "enriched" structural congruence relation.

To introduce *up–to bisimulation* (see [San95]), we consider functions from relations to relations, ranged over by $\mathcal{F}$, and modify the definition of bisimulation:

Definition 5 (Up-to bisimulation) *Given a function $\mathcal{F}$ from relations to relations, we say that a relation $\mathcal{R}$ is a* bisimulation up to $\mathcal{F}$ *iff the property of Definition 3 holds when we replace "$P' \;\mathcal{R}\; Q'$" by "$P' \;\mathcal{F}(\mathcal{R})\; Q'$".*

Intuitively, an up–to bisimulation relation $\mathcal{R}$ is "smaller" than a bisimulation, the gap being filled by function $\mathcal{F}$, that helps building the "future" of processes related in $\mathcal{R}$. This is of interest, as it allows us to reduce the size of the relations we handle for the task of proving bisimilarity results, whenever we have a *correct* function $\mathcal{F}$: a function $\mathcal{F}$ is said to be correct when ($\mathcal{R}$ is a bisimulation up to $\mathcal{F}$) implies ($\mathcal{R}$ relates bisimilar processes). In the context of π–calculus, we shall use the up–to techniques given by the following Proposition:

Proposition 6 ([San95]) *The up to structural congruence, up to restrictions and up to parallel composition proof techniques for π–calculus terms are defined by the following functions over relations, respectively called $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$:*

$$\begin{aligned}
\mathcal{F}_1(\mathcal{R}) &\overset{def}{=} \{(P,Q);\ \exists P_0, Q_0\ s.t.\ P \equiv P_0,\ P_0\mathcal{R}Q_0\ \textit{and}\ Q_0 \equiv Q\};\\
\mathcal{F}_2(\mathcal{R}) &\overset{def}{=} \{(P,Q);\ \exists P_0, Q_0, \vec{x}\ s.t.\ P = (\nu\vec{x})\,P_0,\ Q = (\nu\vec{x})\,Q_0\ \textit{and}\ P_0\mathcal{R}Q_0\};\\
\mathcal{F}_3(\mathcal{R}) &\overset{def}{=} \{(P,Q);\ \exists P_0, Q_0, T\ s.t.\ P = P_0|T,\ Q = Q_0|T\ \textit{and}\ P_0\mathcal{R}Q_0\}.
\end{aligned}$$

Functions $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ correspond to correct *up–to proof techniques, i.e. proving that $\mathcal{R}$ is a bisimulation up to $\mathcal{F}$ (where $\mathcal{F}$ is one of the aforementioned functions) is sufficient to prove that $\mathcal{R}$ is included in bisimilarity. Moreover, these functions can be combined together, still yielding a correct technique.*

Note that $\mathcal{F}_1$ corresponds to a restriction of the general bisimulation up to bisimilarity technique (where $\sim$ replaces $\equiv$ in the definition of $\mathcal{F}_1$).

2 Automatising Up–To Bisimulation

We discuss here the questions related to the automation of the up–to proof techniques for bisimulation. As stated before, the improvements with respect to [Hir98] are twofold: firstly, a narrower relation is handled in the bisimulation up to bisimilarity proof technique, and secondly, the general up–to bisimulation checking algorithm is modified in order to perform breadth–first search (and to handle *expansion* as well: see Section 4).

2.1 Deciding Up–To Closure

The cornerstone of our method is the ability to decide whether a pair of processes belongs to the closure of a relation up to some correct proof technique; in the sequel, we concentrate on our most powerful proof technique, namely the up to (extended) structural congruence, up to restrictions and up to parallel composition proof technique, embedded by function $\mathcal{F}_3 \circ \mathcal{F}_2 \circ \mathcal{F}_1$ (see Proposition 6 above), where $\circ$ denotes function composition.

Proposition 7 ([Hir98]) *Given a pair (P, Q) of processes and a relation $\mathcal{R}$, we can decide whether $(P, Q) \in \mathcal{F}_3 \circ \mathcal{F}_2 \circ \mathcal{F}_1(\mathcal{R})$.*

The proof given in [Hir98] is straightforwardly adapted to handle extended structural congruence as defined above. Let us now explain how we exploit this result for up–to bisimulation verification.

2.2 The General Checking Method

Within the framework of bisimulation verification, an immediate though determining remark is that functions $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}_3$ are *syntactical* operators; this compells us to use an "on the fly" algorithm for bisimulation checking, in order to have access to the actual term corresponding to a given state in the verification process (as opposed for example to the partition refinement algorithm [PT87], which works on *unfoldings* of processes). The "classical" on the fly checking method [FM91] can easily be adapted to the up–to methods, so as to define a semi–decision procedure for up–to bisimulation verification in the π–calculus, as is done in [Hir98]. However, to exploit the up–to methods in full effect, we need to modify the general bisimulation verification algorithm; this is the subject of the remainder of this Section.

Indeed, one of the crucial improvements brought by the up–to techniques is the ability, through the up to parallel composition proof technique, to handle in some cases *replicated terms*, i.e. terms possibly having an infinite behaviour. Thus, when applied, this proof method can allow one to cut infinite branches in the state space of the processes being compared. Using depth–first search, as the original algorithm of [FM91] does, it can be the case that we miss two matching branches which can be cut using the up to parallel composition technique, and instead enter an infinite loop (corresponding to an infinite growth of the state space). To avoid this, it is preferable to adopt a breadth–first strategy, in order to enter an infinite loop only when compulsory. Figure 2 presents the overall behaviour of Mounier's original "on the fly" algorithm, that progressively builds a candidate bisimulation relation, by exploring the states which can be reached starting from two processes.

Let us explain informally how the verification method works (a more detailed description can be found in [FM91] and in the author's forthcoming PhD thesis). The algorithm explores the state space given by the cartesian product of the transition systems induced by the two processes being checked. The data structures involved in the algorithm are: a structure $\mathbb{S}$ (we shall return later on to the nature of $\mathbb{S}$), that contains the states that still have

```
W:= $\emptyset$;
($*$)  $\mathcal{R}$={$(P, Q)$}, V:= $\emptyset$, R:= $\emptyset$, status:=true; insert $(P, Q)$ in $\mathbb{S}$;
while $\mathbb{S}$ is not empty do
  choose a pair $(P_0, Q_0)$ in $\mathbb{S}$ and remove it from $\mathbb{S}$;
  if $(P_0, Q_0)$ **succeeds**
              then (add $(P_0, Q_0)$ to V (or R, see below); **propagate**)
              else
  if $(P_0, Q_0)$ **fails** then
    if $(P_0, Q_0) \in R$
      then (remove $(P_0, Q_0)$ from V and insert it in W,
            status:=false; **propagate**)
      else (insert $(P_0, Q_0)$ in W; **propagate**);
  else (* neither succeeds nor fails *)
    compute the successors of $(P_0, Q_0)$ and insert them in $\mathbb{S}$;
endwhile;
if $P \sim Q$ then (if status then true else loop back to ($*$) ) else false
```

Figure 2: General "On The Fly" Bisimulation Checking Algorithm

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 are discovered 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. a pair in R is actually made of non–bisimilar processes): to handle this, a boolean flag called `status` records the "reliability" of the current computation. When a computation gives an unreliable result, we start all over again, keeping track in W of the states that have been proved non bisimilar.

We still have to explain the behaviour of functions **succeeds**, **fails**, and **propagate**: function **fails** tests if a failure can be detected by inspecting the immediate transitions of the two processes to be examined (a failure occurs if one process performs an action that cannot be done by the other term), or if the two processes are known to be non–bisimilar (set W). Symmetrically, **succeeds** checks if the two processes are trivially bisimilar (i.e. if they are α–convertible or both have no transition) or if there is an assumption stating bisimilarity between them (sets V and R). Finally, function **propagate** propagates a newly found information along the state space, possibly discovering new (non–)bisimilarity properties, or cutting branches that do not have to be explored anymore.

As expected, the nature of the structure $\mathbb{S}$ determines the exploring strategy: with respect to Mounier's algorithm, we replace the *stack* of states that have to be examined by a *queue*, thus performing breadth–first instead of depth–first search. Some extra information regarding the representation of the "vertical" structure of the state space (i.e. the relationships between states due to the transition relation), that comes for free in the case of a stack structure, has to be provided within the objects that are stored in $\mathbb{S}$[1]. Using breadth–first search, we get a *computationally complete* checking algorithm:

Theorem 8 *We say that a bisimulation checking algorithm is* computationally complete *with respect to a given up–to technique $\mathcal{F}$ whenever it diverges if and only no finite bisimulation up to $\mathcal{F}$ relating the two processes to be checked and derivatives of these processes exists.*

With this terminology, the breadth–first version of the algorithm of Figure 2 is computationally complete with respect to the up to structural congruence, up to restrictions and up to parallel composition proof technique.

The proof of this result follows immediately from the definition of the breadth–first search. Note that the hypothesis abouth the relation containing only *derivatives* of the processes we examine is determinant for the proof. What is really important here is the form of completeness we get: while for finite–state processes, depth–first and breadth–first searches differ only "strategically", in the case of replicated terms, the breadth–first version, in conjunction with the up to parallel composition proof technique, provides a real gain in expressiveness, as expressed by Theorem 8. This improvement of the system, together with the additional structural congruence law seen above, will be determinant for the treatment of the examples in Sections 3 and 4.

2.3 A Tool for Up–To Bisimulation Verification

The methods presented so far are implemented in a tool, written in O'Caml, 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

[1]This structural information is even more intricate in the weak case, where some care has to be provided to avoid τ–loops before an action actually fires (i.e. in the "first $\Rightarrow$ part" of $\Rightarrow\xrightarrow{\mu}\Rightarrow$: see Section 4).

up–to proof techniques and between strong bisimulation and expansion (see below for the definition of this notion). Facilities for interactively simulating the behaviour of a process and for debugging (in the case of a check failure) are also provided. Internally, the system works systematically up to structural congruence, i.e. with *normalised* terms, as defined in [Hir98]; as the algorithm of Figure 2 is run, information can be obtained on the size of the current state space to be explored. The tool is available at `http://cermics.enpc.fr/~dh/pi`.

3 First Case Study: Persistent Values and Sharing

One of the applications of the up to parallel composition proof technique is the proof of the so–called *replication theorems*, which express some properties of processes that model resources. Intuitively, a term of the form $(\nu a)\,(!a(\vec{b}).P \mid Q)$ is viewed as an agent Q having access to a private resource P, located at channel a, with the possibility of instantiating the resource with some parameters $\vec{b}$. The resource is usually replicated, because it is meant to be "always available". Resource processes of the form $!a(\vec{b}).P$ are ubiquitous in the study of π–calculus: they can be found in particular in the encodings of the λ–calculus, where application is represented as (the translation of) the function having access to its argument as a resource. Resource processes arise also in the study of higher–order π–calculus, of object calculus, and (as can be expected) of data structures. We focus here on the latter subject, by studying Milner's encoding of lists in the π–calculus.

In [Mil91], lists are represented using two kinds of π–calculus terms, corresponding to each constructor: a process of the form $l(c,n).\overline{c}[v;l']$ represents a *Cons* node, situated at l, and containing a value that can be accessed at channel v and a reference to the remainder of the list situated at l', while $l(c,n).\overline{n}$ is the empty list *Nil* situated at l. Lists can be interrogated by sending two names at their location, one for each possible constructor. With this representation, for example, list $L = [\mathsf{true};\mathsf{false}]$ is defined by the following term, given an encoding of the booleans true and false[2]:

$$L \overset{def}{=} (\nu l_1, b_1)\,(l_0(c,n).\overline{c}[b_1;l_1] \mid b_1.\mathsf{true} \\ \mid (\nu l_2, b_2)\,(l_1(c,n).\overline{c}[b_2;l_2] \mid b_2.\mathsf{false} \mid l_2(c,n).\overline{n}))\,.$$

[2]Along the lines of the encoding of booleans in the λ–calculus, the value true located at b, written $b.\mathsf{true}$, is a process waiting for two names t and f, and returning a signal on t (false would return a signal on f): we write $b.\mathsf{true} \overset{def}{=} b(t,f).\overline{t}$ (see [Mil91]).

Such a data structure, though, is linear: reading the values in a list destroys it. To make the structure persistent, we use replication. *A priori*, there are two approaches to achieve this, whether we choose to replicate the nodes and the value cells in the lists, or the subcomponents; in the case of list L above, this leads to the two following terms:

$$\begin{aligned} L_1 &\overset{def}{=} (\nu l_1, b_1)\,(!l_0(c,n).\overline{c}[b_1;l_1] \mid !b_1.\mathsf{true} \\ &\qquad\qquad \mid (\nu l_2, b_2)\,(!l_1(c,n).\overline{c}[b_2;l_2] \mid !b_2.\mathsf{false} \mid l_2(c,n).\overline{n}))\,; \\ L_2 &\overset{def}{=} !l_0(c,n).(\nu l_1, b_1)\,(\overline{c}[b_1;l_1] \mid !b_1.\mathsf{true} \\ &\qquad\qquad \mid !l_1(c,n).(\nu l_2, b_2)\,(\overline{c}[b_2;l_2] \mid !b_2.\mathsf{false} \mid !l_2(c,n).\overline{n}))\,. \end{aligned}$$

As we shall see, this design choice gets reflected both on the behaviour of the corresponding data structure and on questions related to bisimulation checking.

In the following, to illustrate the behaviour of replicated data, we shall work with a very simple list, consisting only in the element true; as an ephemeral data structure, it is encoded by the following process:

$$R_0 \overset{def}{=} (\nu b, l_1)\,(l_0(c.n)\,\overline{c}[b;l_1] \mid !b.\mathsf{true} \mid l_1(c,n).\overline{n})\,.$$

We simplify this representation by taking $R \overset{def}{=} (\nu b)\,(l_0(c).\overline{c}b \mid !b\,\mathsf{true})$. This process can be viewed as a single constant cell holding the value true, and is only reminiscent of the list $[\mathsf{true}]$; however, it will be sufficent for our task, and the reasoning made below holds for "real" lists as well.

Along the lines exposed above, there are two ways to transform R into a persistent value, represented by the following two processes:

$$R_1 \overset{def}{=} (\nu b)\,(!l_0(c).\overline{c} \mid !b.\mathsf{true}) \qquad \text{and} \qquad R_2 \overset{def}{=} !l_0(c).(\nu b)\,(\overline{c}b \mid !b.\mathsf{true})\,.$$

As stated in [Mil91], the difference between R_1 and R_2 is reflected on *sharing properties* of our data structures. The problem of value sharing can be expressed in terms of security: suppose we have an agent A, willing to interrogate any number of times the resource R located at l_0, and to send a signal on either one of two channels o and p, depending on the value read at l_0. This agent can be represented by the process

$$A \overset{def}{=} !\overline{l_0}c_1.c_1(b).(\nu t, f)\,(\overline{b}[t,f] \mid t.\overline{o} \mid f.\overline{p})\,,$$

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

$$E \overset{def}{=} \overline{l_0}c_2.c_2(b).b(t,f)\overline{f}\,.$$

Our whole system is made of the parallel composition of the resource (being either R_1 or R_2), "innocent" agent A, and "evil" agent E:

$$\mathrm{Sys}_i \overset{def}{=} (\nu l_0)\,(\,R_i \mid A \mid E\,) \qquad i = 1, 2\,.$$

We verify that, depending on which representation we take for the persistent cell R, E can either disturb the communications performed by A, or be innocuous. To this end, we compare the behaviour of the term Sys_i for $i = 1, 2$ with the behaviour of the process where each agent A and E have their own private copy of the resource, corresponding to the following definition:

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

Indeed, if we run the tool with these definitions, we get the following results:

- **case $\mathbf{Sys}_1$:** in this case, a location b for the value true is created once for all and transmitted every process that interrogates the list 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. We check this on a simple session of our tool; we first define the two components of the pair of processes to be checked:

```
> Left (^l0)(^c1)(^c2)(((^b)(!l0(c).c[b]|b(t;f).t[]))
  | l0[c1].c1(b).b(t;f).f[]
  | l0[c2].c2(b).(^t)(^f)(b[t;f]|t.o[]|f.p[]))

> Right (^l0)(^c1)(((^b)(!l0(c).c[b]|b(t;f).t[]))
    | l0[c1].c1(b).b(t;f).f[])
  | (^l0)(^c2)(((^b)(!l0(c).c[b]|b(t;f).t[]))
    | l0[c2].c2(b).(^t)(^f)(b[t;f]|t.o[]|f.p[]))
```

  The syntax for processes is quite natural: ^ stands for abstraction, [] for output and () for input. We now check for bisimilarity:

```
> Check
The processes 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 list, we can prove that Sys_2 and Sys'_2 are bisimilar:

  ```
  > Left (^l0)(((!l0(c).(^b)c[b].b(t;f).t[]))
    | l0[c1].c1(b).b(t;f).f[]
    | !l0[c2].c2(b).(^t)(^f)(b[t;f]|t.o[]|f.p[]))

  > Right (^l0)(((!l0(c).(^b)c[b].b(t;f).t[]))
      | l0[c1].c1(b).b(t;f).f[])
    | (^l0)(((!l0(c).(^b)c[b].b(t;f).t[]))
      | !l0[c2].c2(b).(^t)(^f)(b[t;f]|t.o[]|f.p[]))

  > Check
  The processes are bisimilar.
  ```

  When the tool performs this verification, it uses the up to parallel composition proof technique to erase copies of the cell each time they appear in both process: an infinite proof is thus replaced by a finite one.

The latter example exploits the up to parallel composition proof technique to handle the representation of data structures in the π–calculus. In this context, our tool has been useful in dealing with non trivial properties of processes, and it should be stressed that other existing systems cannot handle replicated processes like those we manipulate here.

Furthermore, it should be remarked that security properties like the one we have considered can be adressed using type systems for the π–calculus. As exposed for example in [San97] an easy way to prevent E from "pretending" to be R by locating the value false at b is to forbid processes that interact with R to use b in input subject position (as E does).

4 Second Case Study: the Alternating Bit Protocol

We now focus on another example: the well–known *alternating bit protocol*, which is probably the most widely used benchmark for verification systems in concurrency. Our purpose here is not to study it *per se*, but instead to use it to shed light on issues related to the behaviour of encodings in the π–calculus. To this end, we shall need *expansion* between processes:

Definition 9 (Expansion) *We write $\Rightarrow$ for the reflexive transitive closure of $\xrightarrow{\tau}$, $\overset{\mu}{\Rightarrow}$ for $\Rightarrow\xrightarrow{\mu}\Rightarrow$, and $\xrightarrow{\hat{\mu}}$ for $\xrightarrow{\mu}$ if $\mu \neq \tau$, $=$ or $\xrightarrow{\tau}$ otherwise.*

We say that a relation $\mathcal{R}$ is an expansion *iff, whenever $P\mathcal{R}Q$, $P \xrightarrow{\mu} P'$ implies that there exists a Q' s.t. $Q \overset{\mu}{\Rightarrow} Q'$ and $P'\mathcal{R}Q'$, and $Q \xrightarrow{\mu} Q'$ implies that there exists P' s.t. $P \xrightarrow{\hat{\mu}} P'$ and $P'\mathcal{R}Q'$. In this case, we write $P \lesssim Q$ for $P\mathcal{R}Q$.*

$\lesssim$ is a preorder on processes that is more realistic than (strong) bisimulation for reasoning about larger case–studies such as a protocol. It allows one to prove that a given term "respects" a behaviour, modulo some extra τ steps (given by transitions $\overset{\mu}{\Rightarrow}$): typically, the duality *specification vs. implementation* can be expressed using expansion. Our treatment of the up–to techniques is straightforwardly adapted to handle expansion; the user of our tool can choose between a *strong* and a *weak* mode, corresponding to relations $\sim$ and $\lesssim$ respectively.

We first informally introduce the protocol, then present an encoding adapted from [Mam98], and finally discuss the encodings we use in order to automatically perform the correctness proof of the protocol. Our purpose in the latter part is to show that the question of defining encodings in the framework of verification differs from the usual problem (i.e. the study of *expressiveness*), as exemplified on our case study.

4.1 The Protocol

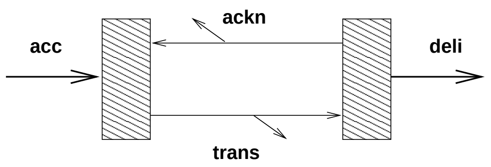

Figure 3: The Alternating Bit Protocol

Figure 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, a boolean tag is associated 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 boolean to the first agent on ackn, as an acknowledgment. Channel ackn being lossy as well, the second agent has to repeatedly send on it. When the acknowledgment finally arrives, a new cycle can start with the negated boolean, so that the first agent can tell when the message has actually been received.

The π–calculus processes[3] implementing this protocol are defined on Figure 4 (again, our encoding is adapted from [Mam98]): the first agent's behaviour is given by processes *Send* and *Wait*, while the second agent corresponds to process *Receive*; message losses on trans and ackn are managed by process *Noise*, that sends a signal on a channel loss whenever a signal gets lost (one could see these synchronisations on loss as a representation of a timer mechanism to detect losses). Note that the agents are parametrised upon two booleans: this will be discussed below. 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 c.

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

Figure 4: Modelling the Alternating Bit Protocol

[3]The language we use here is actually value–passing CCS; for the encodings discussed below, however, we shall need the full power of π–calculus.

4.2 The Encodings – Discussion

Choice – To model the alternating bit protocol in our system, we have to deal with two constructs that are not present in our language, namely operators on booleans and the choice operator. Well–known encodings of these constructs exist (see [Mil91, Nes97]); these encodings belong to theoretical studies, and have been mechanised e.g. in the programming language PICT [PT97]. We shall see here the problems that arise as we try to use them in the context of verification, and how they can be treated.

Various encodings of guarded choice are investigated 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 comitted. In general, once a choice has been taken, the "dead" branches remain, and, depending on the encoding we choose, are "more or less alive"; when it comes to automatic verification, we need to be able to erase these branches *syntactically*, to prevent our relation from growing. We have chosen to encode the binary choice of Figure 4 as follows ($[\![\;]\!]$ represents the encoding function):

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

This encoding represents an adaptation of those described in [NP96]; here, a lock channel l is created and when one branch is chosen, it deactivates the other branch by consuming its head prefix, thus leading to a subterm of the form $(\nu l)\,(l.T)$, that can immediately be garbage collected using the additional law for structural congruence (see Section 1). Our simplified encoding is correct due to an important property of the terms we consider: each time a choice construct is encountered in a run of the protocol (agents *Wait* and *Noise*), at most one process can interact with the branches involved in the choice (that is, we cannot have simultaneous emissions on ackn and loss, nor on ackn and trans). Thus, it cannot be the case that both branches commit before the lock l is consumed, and the untaken branch can be safely deactivated before the chosen branch proceeds.

Furthermore, note as well that the simplest encoding of guarded choice, defined as follows

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

and called *internal choice* in [NP96], cannot be adopted here, as the choice of the comitting branch depends on the context (since at most one branch can commit), and cannot be done internally *before* synchronisation.

Note that the original encoding of [NP96] could not be chosen, since it allows the computation to proceed before the dead branches have been deactivated.

Booleans – We have already seen how booleans are encoded in the π–calculus. The encoding presented in [Mam98] uses a single boolean as parameter for the various agents, and the negation operator for the recursive calls in *Send* and *Receive*. We could have encoded the latter operator as

$$[\![\neg b]\!] \overset{def}{=} !b'(t,f)(\nu t',f')\,(\overline{b}[t';f']|t'.\overline{f}|f'.\overline{t})\,; \qquad (*)$$

this encoding is not convenient for the purpose of verification, because each time the protocol loops, it keeps adding new terms corresponding to the negation operator to the system, making the relation grow *ad infinitum* (intuitively, $\neg\neg b$ does not reduce to b). We therefore use a trick, that consists in parametrising all the agents upon *two* booleans, that are exchanged as we want to make a recursive call with the negated bit (see Figure 4).

We also have to manage the testing construct in agent *Receive*: here again, we take advantage of the additional law for structural congruence to get rid of inactive terms once the test performed. We represent the boolean operator "=" (that can be seen as the negation of the XOR operator) by:

$$\begin{array}{rcl} [\![b = b']\!] & \overset{def}{=} & b_0(T,F).\,(\nu t,f,t',f') \\ & & (\overline{b}[t;f] \mid \overline{b'}[t';f'] \mid t.(t'.\overline{T}|f'.\overline{F}) \mid f.(t'.\overline{F}|f'.\overline{T}))\,. \end{array}$$

Once the test has taken place, non–chosen branches are automatically garbage collected in the normalisation process.

Having defined the encoding of the protocol into our simple language, we provide our tool with the corresponding definitions, and the correctness proof of the property $\mathit{Specif} \lesssim \mathit{System}$ is performed automatically:

```
> Left (^c)(!c.acc.deli[].c[] | c[])

> Right (^tra)(^sen)(^wai)(^ack)(^rec)(^b1)(^b2)(^lt)(^loss)(
!sen(b1;b2).acc.tra[b2;b1].wai[b2;b1]
| !wai(b1;b2).(^l)(l[] | ack(bp1;bp2).l.loss[].sen[bp1;bp2]
   | loss.l.ack[b1;b2].lt[].tra[b1;b2].wai[b1;b2])
| !rec(b1;b2).tra(bp1;bp2).(^t)(^f)(^t1)(^f1)(^tp)(^fp)
   (b1[t;f] | bp1[t1;f1] | t.(t1.tp[] | f1.fp[])
   | f.(t1.fp[] | f1.tp[]) | tp.(ack[b1;b2].rec[b1;b2])
   | fp.del[].(ack[b2;b1].rec[b2;b1]))
| !lt.(^lo)(tra(bx;by).lo.ack[bx;by].loss[] |
   ack(bz;bt).lo.tra[bz;bt].loss[] | lo[])
| lt[] | sen[b1;b2] | rec[b1;b2] | !b1(t;f).t[] | !b2(t;f).f[])

> Mode
Checking mode is weak, verbose mode is off,
proof technique is up to restriction up to parallel composition.

> Check
The right process expands the left one.
```

Conclusion

We have seen how the up–to techniques for bisimulation can be adapted for the purpose of verification. The examples of persistent lists and of the proof of the alternating bit protocol suggest that the task of dealing with encodings in the π–calculus can be managed using these techniques. It is questionable, though, whether one should adopt a standpoint analogous to the design choices of PICT [PT97] , where a full-size programming language is built by adding successive layers to a simple core language through encodings of higher–level constructs. Indeed, developping an implementation in a system built this way adds transitions to the terms and calls for clever use of garbage collection (as seen on the examples); such issues are critical in the field of verification. Nevertheless, the experiments shown above are interesting both from a theoretical and from a practical point of view.

From a theoretical point of view, we have seen how the application of theoricians' techniques (as the up–to methods originally are) for the purpose of verification gives extra insight on results such as the encodings of language constructs or of data. A striking example is the straightforward definition of the negation operator for booleans (see equation marked with $(*)$), which is catastrophic in terms of state space growth, as stated above.

From a practical point of view, this work seems encouraging for the use of the up–to techniques in the field of verification. Of course, constructs like booleans or the choice operator should probably belong to a "real size" system. However, the task of specification in a verification tool usually involves some formalisations that, on a larger scale, are quite akin to the encodings we have seen, at least from a methodological point of view. In this approach, the up–to techniques have been shown to be helpful in terms both of efficiency and of expressiveness.

Regarding future work, it could be interesting to study whether the up–to techniques can be integrated in a tool like the Mobility WorkBench [VM94], that is a much richer system than our prototype, and that moreover supports various supplementary facilities. On its own, however, our tool can be enriched to perform some more experiments concerning the verification of π–calculus bisimilarity results; larger case studies, for example, could be considered, or other encodings. Work is also in progress to adapt the up–to bisimulation verification method to *open terms*, i.e. terms possibly containing process variables, in order to be able to prove not only bisimilarity results, but also general *bisimilarity laws*. Finally, the issue of the usefulness of the up–to techniques within other models of concurrency should also be investigated.

References

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

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

[Mam98] B. Mammas. Une mtéthodologie de preuves orientée contraintes et basée sur les systèmes de transitions modales. Technical report, LIP6, UPMC, 1998.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, LFCS, October 1991.

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

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

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

[PT97] B. C. Pierce and D. N. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997.

[San95] D. Sangiorgi. On the bisimulation proof method. In *Proceedings of MFCS '95*, volume 969 of *LNCS*, 1995.

[San96] D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Informatica*, 33:69–97, 1996.

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

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