Translating Pi-Calculus into LOTOS NT

Radu Mateescu[1] and Gwen Salaün[1,2]

[1] Inria Grenoble – Rhône-Alpes / Vasy project-team / Lig, Inovallée
655, av. de l'Europe, Montbonnot, F-38334 Saint Ismier, France
[2] Grenoble Inp, 46, av. Félix Viallet, F-38031 Grenoble, France
{Radu.Mateescu,Gwen.Salaun}@inria.fr

Abstract. Process calculi supporting mobile communication, such as the π-calculus, are often seen as an evolution of classical value-passing calculi, in which communication between processes takes place along a fixed network of static channels. In this paper, we attempt to bring these calculi closer by proposing a translation from the finite control fragment of the π-calculus to Lotos NT, a value-passing concurrent language with classical process algebra flavour. Our translation is succinct in the size of the π-calculus specification and preserves the semantics of the language by ensuring a one-to-one correspondence between the states and transitions of the labeled transition systems corresponding to the input π-calculus and the output Lotos NT specifications. We automated this translation by means of the Pic2Lnt tool, which makes it possible to analyze π-calculus specifications using all the state-of-the-art simulation and verification functionalities provided by the Cadp toolbox.

1 Introduction

Process calculi (or algebras) are abstract specification languages used to model concurrent systems. These formalisms have been widely studied and used for the specification of real-world systems in many different application areas such as telecommunication protocols, hardware design, or embedded systems. One of the most famous calculi is the π-calculus [20] proposed by Milner, Parrow, and Walker about twenty years ago. The π-calculus is an extension of Ccs [18] with mobile communication, and is equipped with an operational semantics defined in terms of labeled transition systems (Ltss). Although a lot of theoretical results have been achieved on this language (see [24] for a survey), only a few verification techniques and tools, such as the Mobility Workbench (Mwb) [28] or Jack [8], are operational for analyzing π-calculus specifications automatically.

In this paper, we attempt to provide similar analysis features for π-calculus specifications by reusing the verification technology already available for classical (*i.e.*, without mobility) value-passing process algebras. Contrary to existing analysis tools for the π-calculus, which rely on specific algorithms and intermediate models, such as HD-automata [8], our approach is based on a novel translation from π-calculus to a classical process algebra. We focus here on the finite control fragment of the π-calculus and adopt as target language Lotos NT [4], a recent enhancement of Lotos [16]. In Lotos NT, the abstract data type part was

abandoned in favor of constructive data type definitions and pattern-matching, and the behavior process algebraic part was replaced by an imperative-like language with a user-friendly syntax. To the best of our knowledge, this is the first π-calculus translation that uses a classical process algebra as target language.

Most of the π-calculus constructs can be translated quite straightforwardly into LOTOS NT thanks to its good level of expressiveness. Nevertheless, we faced some subtle difficulties in order to have a translation as succinct as possible and preserving the LTS semantics, *i.e.*, mapping each transition of a π-calculus agent to a transition of the resulting LOTOS NT term. Obviously, one of the main problems was to emulate mobile communication in a language that offers only communication on static channels; we overcame this issue by heavily exploiting the data types and synchronization features of LOTOS NT. Our translation is fully automated by the PIC2LNT tool we have implemented. Since LOTOS NT is one of the input languages of the CADP [12] verification toolbox, all the state-of-the-art verification features of CADP can be used on the LOTOS NT specifications generated from the π-calculus ones.

The outline of the paper is as follows. In Section 2, we introduce both specification languages, namely the π-calculus and LOTOS NT. Section 3 presents the translation rules and shows the semantics preservation. Section 4 describes the PIC2LNT translator and Section 5 illustrates the overall approach on the specification and verification of a simple Web services case study. Finally, Section 6 compares our proposal with related approaches, and Section 7 draws up some conclusions and lines for future work.

2 Pi-Calculus and LOTOS NT

We briefly present below the syntax and semantics of π-calculus and of the LOTOS NT fragment that serves as target language for the translation.

Pi-Calculus. We consider the original version of π-calculus [20] equipped with the early operational semantics defined in [21]. For simplicity of the presentation, we focus on the monadic π-calculus, although the translation to LOTOS NT given in Section 3 can be straightforwardly extended to handle the polyadic version of the calculus [19]. The syntax and semantics of the π-calculus are shown in Figure 1. Channel names (denoted by $a, ..., z$) belong to an infinite countable set of names $\mathcal{N}$. Agents (denoted by P) are built from inaction (0), action prefix (.), parallel composition (|), choice (+), channel creation (ν), guard ([ ]), and instantiation ($A(...)$). The occurrences of y in $x(y).P$ and $(\nu y)P$ are bound and the other occurrences of channel names are free. The set of free (resp. bound) names of an agent P is denoted by $fn(P)$ (resp. $bn(P)$), and the set of names of P is defined as $n(P) = fn(P) \cup bn(P)$. Each agent identifier A has an arity $r(A) \geq 0$ and must be defined by an equation $A(x_1, ..., x_{r(A)}) \stackrel{\text{def}}{=} P$. The parameter names $x_1, ..., x_{r(A)}$ must be pairwise distinct and $fn(P) \subseteq \{x_1, ..., x_{r(A)}\}$.

The actions (denoted by α) that an agent can perform are of four kinds: internal action (τ), free output ($\overline{x}y$), bound output ($\overline{x}(z)$), and free input (xy). The

$P ::= 0 \mid \tau.P \mid \overline{x}y.P \mid x(y).P \mid P_1|P_2 \mid P_1 + P_2 \mid (\nu x)P \mid [x = y]P \mid [x \neq y]P \mid A(x_1, ..., x_{r(A)})$ $\quad \alpha ::= \tau \mid \overline{x}y \mid \overline{x}(z) \mid xy$

TAU $\tau.P \xrightarrow{\tau} P$ OUT $\overline{x}y.P \xrightarrow{\overline{x}y} P$ IN $x(y).P \xrightarrow{xz} P\{z/y\}$

SUM $\dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1 + P_2 \xrightarrow{\alpha} P_1'}$ PAR $\dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1|P_2 \xrightarrow{\alpha} P_1'|P_2}$ if $bn(\alpha) \cap fn(P_2) = \emptyset$

COM $\dfrac{P_1 \xrightarrow{\overline{x}y} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} P_1'|P_2'}$ CLOSE $\dfrac{P_1 \xrightarrow{\overline{x}(y)} P_1' \quad P_2 \xrightarrow{xy} P_2'}{P_1|P_2 \xrightarrow{\tau} (\nu y)(P_1'|P_2')}$ if $y \notin fn(P_2)$

RES $\dfrac{P \xrightarrow{\alpha} P'}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'}$ if $x \notin n(\alpha)$ OPEN $\dfrac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(z)} P'\{z/y\}}$ if $x \neq y, z \notin fn((\nu y)P')$

MATCH $\dfrac{P \xrightarrow{\alpha} P'}{[x = x]P \xrightarrow{\alpha} P'}$ MISMATCH $\dfrac{P \xrightarrow{\alpha} P'}{[x \neq y]P \xrightarrow{\alpha} P'}$ if $x \neq y$

IDE $\dfrac{P\{y_1/x_1, ..., y_{r(A)}/x_{r(A)}\} \xrightarrow{\alpha} P'}{A(y_1, ..., y_{r(A)}) \xrightarrow{\alpha} P'}$ if $A(x_1, ..., x_{r(A)}) \stackrel{\text{def}}{=} P$

Fig. 1. Syntax and early operational semantics of π-calculus

same notations $fn(\alpha)$, $bn(\alpha)$, and $n(\alpha)$ are used for actions, the only bound occurrence being z in the bound output $\overline{x}(z)$. Intuitively, bound names in actions correspond to references of places in the executing agent where substitutions must be performed; bound outputs are used to represent scope extrusions (rules OPEN and CLOSE). The early operational semantics is given in terms of rules enabling to infer the transitions, labeled by actions, that an agent can perform. The rules associated to binary operators (| and +) have also symmetric forms, omitted here for conciseness. Given a substitution $\sigma : \mathcal{N} \rightarrow \mathcal{N}$, we denote by $P\sigma$ the agent P in which each free name x has been replaced by $\sigma(x)$, possibly with changes of the bound variables to avoid captures. The substitution $\{y_1/x_1, ..., y_n/x_n\}$ maps each x_i into y_i for $i \in [1, n]$ and keeps all the other names unchanged. In the sequel, we will consider only π-calculus agents that satisfy the *finite control* property [6], *i.e.*, do not contain recursive calls of agent identifiers through the parallel composition operator.

LOTOS NT fragment. LOTOS NT [4] is a simplified variant of the E-LOTOS standard [15] that attempts to combine the best features of imperative programming languages and value-passing process algebras. LOTOS NT has a user-friendly syntax and a formal operational semantics defined in terms of labeled transition systems (LTSs). LOTOS NT is supported by the LNT.OPEN tool of CADP [12], which allows the on-the-fly exploration of the LTSs corresponding to LOTOS NT specifications. As target of our translation, we use only a small fragment of LOTOS NT, whose syntax and semantics are given in Figure 2.

LOTOS NT terms (denoted by B) are built from actions, choice ("**select**"), conditional ("**if**"), sequential composition (";"), hiding ("**hide**"), and parallel composition ("**par**"). Communication is carried out by rendezvous on gates G with bidirectional transmission of multiple values (for simplicity, in Fig. 2 we

B ::= **stop** | **null** | $G(!E, ?x)$ **where** E' | $B_1; B_2$ | **if** E **then** B **end if**
| **var** $x{:}T$ **in** $x := E; B$ **end var** | **hide** G **in** B **end hide**
| **select** [**var** $x_1{:}T_1, ..., x_n{:}T_n$ **in**] B_1 [] ... [] B_n **end select**
| **par** G **in** $B_1||...||B_n$ **end par** | $P[g_1, ..., g_m](E_1, ..., E_n)$

LNT-NULL **null** $\xrightarrow{\delta}$ **stop**

LNT-ACT $\dfrac{v' \in type(x) \wedge [\![E'\{v'/x\}]\!] = \text{true}}{G(!E, ?x) \text{ where } E'; B \xrightarrow{G\ ![\![E]\!]\ !v'} B\{v'/x\}}$

LNT-SEQ-1 $\dfrac{B_1 \xrightarrow{\beta} B_1'}{B_1; B_2 \xrightarrow{\beta} B_1'; B_2}$

LNT-SEQ-2 $\dfrac{B_1 \xrightarrow{\delta} B_1' \quad B_2 \xrightarrow{\beta} B_2'}{B_1; B_2 \xrightarrow{\beta} B_2'}$

LNT-IF $\dfrac{[\![E]\!] = \text{true} \quad B \xrightarrow{\beta} B'}{\textbf{if } E \textbf{ then } B \textbf{ end if} \xrightarrow{\beta} B'}$

LNT-VAR $\dfrac{B\{[\![E]\!]/x\} \xrightarrow{\beta} B'}{\textbf{var } x{:}T \textbf{ in } x := E; B \textbf{ end var} \xrightarrow{\beta} B'}$

LNT-HID-1 $\dfrac{B \xrightarrow{\beta} B' \quad gate(\beta) \neq G}{\textbf{hide } G \textbf{ in } B \textbf{ end hide} \xrightarrow{\beta} \textbf{hide } G \textbf{ in } B' \textbf{ end hide}}$

LNT-HID-2 $\dfrac{B \xrightarrow{\beta} B' \quad gate(\beta) = G}{\textbf{hide } G \textbf{ in } B \textbf{ end hide} \xrightarrow{\mathtt{i}} \textbf{hide } G \textbf{ in } B' \textbf{ end hide}}$

LNT-SEL $\dfrac{i \in [1, n] \quad B_i \xrightarrow{\beta} B_i'}{\textbf{select } [\textbf{var } x_1{:}T_1, ..., x_n{:}T_n \textbf{ in}]\ B_1 [] ... [] B_n \textbf{ end select} \xrightarrow{\beta} B_i'}$

LNT-PAR $\dfrac{i \in [1, n] \quad B_i \xrightarrow{\beta} B_i' \quad gate(\beta) \neq G}{\textbf{par } G \textbf{ in } B_1||...||B_n \textbf{ end par} \xrightarrow{\beta} \textbf{par } G \textbf{ in } B_1||...||B_i'||...||B_n \textbf{ end par}}$

LNT-COM $\dfrac{I \subseteq [1, n] \quad \forall i \in I. B_i \xrightarrow{\beta} B_i' \quad gate(\beta) = G \quad j \in I}{\textbf{par } G \textbf{ in } B_1||...||B_n \textbf{ end par} \xrightarrow{\beta} \textbf{par } G \textbf{ in } B_1||...||B_j'||...||B_n \textbf{ end par}}$

LNT-IDE $\dfrac{B\{g_1/G_1, ..., g_m/G_m\}\{[\![E_1]\!]/x_1, ..., [\![E_n]\!]/x_n\} \xrightarrow{\beta} B'}{P[g_1, ..., g_m](E_1, ..., E_n) \xrightarrow{\beta} B'}$

where **process** $P[G_1, ..., G_m](x_1{:}T_1, ..., x_n{:}T_n)$ **is** B **end process**

Fig. 2. Syntax and early operational semantics of the LOTOS NT fragment

considered actions with only two values being sent in both directions). Synchronizations may also contain optional guards ("**where**") expressing boolean conditions on received values. The gate on which an action β takes place is denoted by $gate(\beta)$. The special action δ is used for defining the semantics of sequential composition. An action $G(...)$ can occur in isolation, in which case it is considered to be equivalent to $G(...)$; **null**. The internal action is denoted by the special gate `i`, which cannot be used for synchronization. The parallel composition operator allows multiway rendezvous on the same gate. As in LOTOS [16], processes are parameterized by gates and data variables.

The reader familiar with LOTOS may notice that the LOTOS NT fragment considered is not far from LOTOS itself, which could also serve as the target language for the translation. However, as it will become clear in Section 3, LOTOS NT presents at least two advantages *w.r.t.* LOTOS for translating π-calculus agents: (a) the symmetric sequential composition operator ";" of LOTOS NT makes it possible to group together the behaviour following the

branches of a "**select**" statement, thus enabling a succinct translation of nested action prefixes occurring in π-calculus agents, and (b) as opposed to the sequential composition operator ">>" of LOTOS, the semantics of the ";" operator of LOTOS NT does not create spurious internal actions in the LTS, making possible to achieve a one-to-one correspondence between the transitions of a π-calculus agent and those of the LOTOS NT term resulting after translation.

3 Translation from Pi-Calculus to LOTOS NT

The translation presented below maps each π-calculus agent P to a LOTOS NT behaviour term $t(P, \overline{G}, k)$, where $\overline{G}$ is the set of LOTOS NT gates on which the term communicates with its environment and $k \geq 1$ is a natural number identifying the corresponding concurrent activity (*i.e.*, the operand of the immediately enclosing parallel composition operator, if any, which contains the term). Two classes of channels are distinguished: *public* channels correspond to the free channel names occurring in P, whereas *private* channels correspond to channel names bound by ν operators occurring in P. The set $\overline{G}$ includes two predefined gates G_{pub} and G_{priv}, which serve to model the non-synchronized communications on public and private channels, respectively.

Channel names. Since the communication in LOTOS NT takes place along static gates, we cannot use directly these gates to represent mobile communication. Instead, we represent π-calculus channel names as values of a LOTOS NT data type Chan, and we model channel mobility between π-calculus agents by communicating values of this type along gates between the corresponding LOTOS NT processes. The example below shows the definition of the LOTOS NT type Chan for the π-calculus agent $(\nu x)(\overline{a}b.\overline{c}x.0)$.

```
type Chan is
  a, b, c, x (id:Nat) with "==", "!="
end type
function new_id () : Nat is
  !external null
end function
```

```
function is_public (ch:Chan) : Bool is
  case ch in
    a | b | c -> return true
    | any -> return false
  end case
end function
```

The Chan type is equipped with the comparison operators "==" and "!=". It provides a constant constructor for each public channel and a constructor parameterized by a natural number id for each private channel. The predicate is_public characterizes public channels. To create new Chan values when a ν operator is evaluated, we use a function new_id defined externally in C, which returns a new natural number at each invocation.

Inaction and action prefix. The null π-calculus agent 0 is naturally translated into the **stop** LOTOS NT operator, which does not perform any action. The prefix operator is translated using the choice operator "**select**" and the sequential composition operator ";" as shown below. In order to capture all potential interactions that may become possible during execution due to mobility of channels, the communication on a channel x is modeled by a nondeterministic choice on all the gates connecting the current LOTOS NT term to its environment.

Binary synchronizations between the current term and its environment are enforced by emitting the value x of type Chan corresponding to x and the identifier k of the current term, which acts as sender (resp. receiver) for output (resp. input) actions. The semantics of LOTOS NT parallel composition (used to translate the | operator, see below) ensures that only the terms corresponding to different concurrent activities (having different identifiers k) and sharing the same value x can communicate in an unidirectional manner by transmitting a value y of type Chan on some gate. Variables s and r act as placeholders for the identifiers of the sender and receiver terms, respectively.

$t(\overline{x}y.P, \{G_1, ..., G_n, G_{pub}, G_{priv}\}, k) =$	$t(x(y).P, \{G_1, ..., G_n, G_{pub}, G_{priv}\}, k) =$
select var r:Nat **in**	**select var** s:Nat, y:Chan **in**
G_1 (!x, !y, !k, ?r) [] ...	G_1 (!x, ?y, ?s, !k) [] ...
G_n (!x, !y, !k, ?r) []	G_n (!x, ?y, ?s, !k) []
G_{pub} (!x, !y, !true)	G_{pub} (!x, ?y, !false)
where is_public (x) []	**where** is_public (x) []
G_{priv} (!x, !y, !true)	G_{priv} (!x, ?y, !false)
where not (is_public (x))	**where** not (is_public (x))
end select ;	**end select** ;
$t(P, \{G_1, ..., G_n, G_{pub}, G_{priv}\}, k)$	$t(P, \{G_1, ..., G_n, G_{pub}, G_{priv}\}, k)$

When x denotes a public (resp. private) channel, an action on gate G_{pub} (resp. G_{priv}) is added in order to model the possibly non-synchronized execution of send/receive actions, which comprise the emission of a true/false boolean value in order to differentiate them in the LTS of the resulting LOTOS NT term. Note that the translation makes no difference between free and bound output, these actions being distinguished by the value y of type Chan being sent, which can be either public or private.

Sum, match, and mismatch. The sum operator is naturally translated by using the choice operator of LOTOS NT. The match and mismatch operators are translated in terms of the conditional operator.

$t(P_1 + P_2, \overline{G}, k) =$ **select** $t(P_1, \overline{G}, k)$ [] $t(P_2, \overline{G}, k)$ **end select**
$t([x = y]P, \overline{G}, k) =$ **if** x == y **then** $t(P, \overline{G}, k)$ **end if**
$t([x \neq y]P, \overline{G}, k) =$ **if** x != y **then** $t(P, \overline{G}, k)$ **end if**

Note that in the translation of the operands P_1, P_2, and P the set of gates $\overline{G}$ and the identifier k of the current term do not change, since the sum operator is sequential (*i.e.*, it does not create new concurrent activities).

Parallel composition. The parallel composition operator is translated using the "**par**" operator of LOTOS NT. A fresh gate G_{new} is introduced to model the communication between the two LOTOS NT terms resulting from the translation of the operands P_1 and P_2. Since the parallel operator creates two concurrent activities, two new distinct identifiers $2k$ and $2k + 1$ are assigned to the corresponding LOTOS NT terms. Given that G_{new} is added to the sets of gates connecting the two terms to their environment, every send/receive communication carried out by $P_1|P_2$ will also be executed by the whole LOTOS NT term. Indeed,

according to the translation of the action prefix (see above), all input/output operations of P_1 and P_2 will also occur in the two LOTOS NT terms as actions along G_{new}, and the "**par**" operator will enforce their proper synchronization[3]. All synchronizations on G_{new} are renamed into the internal action `i` using the "**hide**" operator to reflect the semantics of the π-calculus communication.

$$t(P_1|P_2, \overline{G}, k) = \textbf{hide } G_{new} \textbf{ in par } G_{new} \textbf{ in } t(P_1, \overline{G} \cup \{G_{new}\}, 2k) \;||\; t(P_2, \overline{G} \cup \{G_{new}\}, 2k+1) \textbf{ end par end hide}$$

The scheme for assigning concurrent activity identifiers yields a contiguous numbering if the direct nestings of parallel operators in the π-calculus agents are arranged to form balanced binary trees. Given that the parallel operator is associative, this can be easily obtained by an adequate insertion of parentheses, *e.g.*, $((P_1|P_2)|(P_3|P_4))$ instead of $(((P_1|P_2)|P_3)|P_4)$, which would be the default parsing of the agent in absence of parentheses.

Channel creation. The channel creation operator (νx) is translated by creating a new private value of type `Chan` and storing it in a variable `x`, which can be subsequently used by the LOTOS NT term.

$$t((\nu x)P, \overline{G}, k) = \textbf{var } \text{x:Chan } \textbf{in } \text{x} := \text{x (new\_id ())}; t(P, \overline{G}, k) \textbf{ end var}$$

This translation rule does not directly forbid the LOTOS NT term to perform an emission along the channel `x`, in the sense that some action $\overline{x}a$ present in P (whose execution is forbidden by the rule RES in Fig. 1) will be translated as an action "G (!x, !a, !k, ?r)" on some gate $G \in \overline{G}$. Such emissions are forbidden indirectly by the way in which action prefix and parallel composition are translated (see above). Indeed, for the synchronization on G to take place, the environment must propose on G an action containing the same fresh value `x`. This is impossible unless `x` has been previously sent by the current LOTOS NT term to the environment, by an emission corresponding to a bound output previously executed by the agent. Thus, scope extrusions are modeled by the communication of fresh values of type `Chan`, which can be subsequently used by different LOTOS NT terms for communication.

Agent definition and instantiation. Agent definitions A are mapped to LOTOS NT process definitions as shown below. In addition to the channel names x_1, ..., $x_{r(A)}$ (represented as values of type `Chan`), the process is parameterized by the gate set $\overline{G}$ and the identifier k, which capture the context of the call.

$$t(A(x_1, ..., x_{r(A)}) \stackrel{\text{def}}{=} P, \overline{G}, k) = \textbf{process } \text{A}_d\ [\overline{G}]\ (x_1, ..., x_{r(A)}\text{:Chan}, k\text{:Nat}) \textbf{ is } t(P, \overline{G}, k) \textbf{ end process}$$
$$t(A(y_1, ..., y_{r(A)}), \overline{G}, k) = \text{A}_d\ [\overline{G}]\ (y_1, ..., y_{r(A)}, k)$$

[3] The translation of an agent $P_1|...|P_n$ requires $n-1$ gates, one for each | operator. The number of gates could be reduced to 1 by using the generalized parallel composition operator (not yet fully implemented in LOTOS NT) proposed in [13], which can model binary synchronization between n processes.

Since an agent identifier may be invoked at several places in the π-calculus agent under translation, and LOTOS NT processes have a fixed number of gate parameters, we chose to produce one LOTOS NT process definition A_d for each occurrence of the agent identifier A in a context where $|\overline{G}| = d$. This increases the size of the LOTOS NT specification by only a logarithmic factor *w.r.t.* the size of the input π-calculus specification (see the discussion on complexity below). Finally, the restriction to finite control agents ensures that all (direct or transitive) recursive invocations of the agent identifier A inside its body P will occur inside the same concurrent activity, and therefore all the corresponding calls to process A_d will have the same context $\overline{G}$ and k. This enables to translate each π-calculus agent definition into a finite number of LOTOS NT processes.

Pi-calculus specification. The π-calculus agent occurring at the top-level of a specification is translated in a context consisting of the gates G_{pub} and G_{priv} (which model the communications on public and private channels, respectively) and a concurrent activity with identifier 1.

$$\pi 2lnt(P) = \textbf{par } G_{priv} \textbf{ in } t(P, \{G_{pub}, G_{priv}\}, 1) \mid\mid \textbf{stop end par}$$

The translation of action prefix and channel creation operator (see above) produces extra synchronizations on gate G_{priv}, which must be forbidden in order to reflect that the environment of the π-calculus agent is not aware of the private channels of P. This is done by a synchronization on G_{priv} with "**stop**", the simplest environment not aware of private channels, added at the top-level of the resulting LOTOS NT term.

Correctness and complexity of the translation. While devising the translation, we sought to preserve the behaviour by ensuring a one-to-one correspondence between the transitions performed by the π-calculus agent and those performed by the LOTOS NT term. The actions α of the agent are related to LOTOS NT actions by the function $h(\alpha, G, k)$, where G is a gate name and $k \geq 1$:

α	$h(\alpha, G, k)$
τ	i
$\overline{x}y$	G !x !y !k ?r:Nat
$\overline{x}(z)$	G !x !z !k ?r:Nat where $\neg$is_public(z)
xy	G !x ?y:Chan ?s:Nat !k

We also define the set $C_k \stackrel{\text{def}}{=} \{2^l \cdot k + r \mid l \geq 0 \wedge r < 2^l\}$, which represents the set of concurrent activity identifiers generated as children of activity k. The following proposition states the correctness of the translation.

Proposition 1 (Behaviour preservation). *Let P be a π-calculus agent, $\overline{G}$ a set of* LOTOS NT *gates, and $k \geq 1$. Then, for every action α and agent P':*

$$P \stackrel{\alpha}{\rightarrow} P' \quad \textit{iff} \quad \exists k' \in C_k \,.\, \forall G \in \overline{G} \,.\, t(P, \overline{G}, k) \stackrel{h(\alpha,G,k')}{\rightarrow} t(P', \overline{G}, k).$$

The proof of this proposition (omitted here due to space limitations) is by induction on the depth of the derivation leading to the transition $P \stackrel{\alpha}{\rightarrow} P'$. When

translating a top-level π-calculus agent P in the context given by $\{G_{pub}, G_{priv}\}$ and activity identifier 1 (see the rule for π-calculus specification), Proposition 1 and the semantics of the "**par**" operator ensure that each internal transition of P is mapped to one internal transition of the LOTOS NT term and each communication on a public channel corresponds to an action on gate G_{pub}. On the other hand, since the synchronizations on G_{priv} are blocked, there are no communications between P and its environment along the private channels of P. Thus, every action performed by a top-level π-calculus agent is mapped into a single action (either `i`, or an action on G_{pub}) of the resulting LOTOS NT term.

In order to estimate the complexity of the translation, we calculate the size of the output LOTOS NT term *w.r.t.* the size $|P|$ of the input π-calculus agent P. The size is defined as the number of operators contained in the LOTOS NT term and in the π-calculus agent, respectively. The definition of type Chan has a size linear *w.r.t.* $|P|$ and each translation rule given above invokes the translation t only once for each operand of P. The only sources of increase in size are the translation rules for action prefixes and for agent definitions. In the worst case, the former rule expands each action of P into $|\overline{G}|_{max}$ actions, and the latter duplicates the definition of an agent as many times as $|\overline{G}|_{max}$, the maximum size of the set $\overline{G}$. Assuming that all nestings of parallel operators inside P are arranged to form balanced binary trees, $|\overline{G}|_{max}$ is bounded by $O(\log |P|)$, which makes the size of the whole LOTOS NT term proportional to $O(|P| \cdot \log |P|)$.

4 Tool Support: Pic2Lnt

We developed an automatic translator tool from π-calculus to LOTOS NT, named PIC2LNT, implemented using the SYNTAX+TRAIAN compiler construction technology [11]. It consists of about 900 lines of SYNTAX code, 2,300 lines of LOTOS NT code, and 500 lines of C code. Although the π-calculus version used in Sections 2 and 3 to illustrate the translation is monadic, the PIC2LNT translator implements a polyadic version of the π-calculus, by exploiting the fact that LOTOS NT allows the communication of multiple values on the same gate. The concrete syntax of polyadic π-calculus accepted by PIC2LNT subsumes the syntax implemented in MWB, with the restriction to finite control agents. Figure 3 gives an overview of the tool chain that makes possible the verification of π-calculus specifications using the CADP toolbox [12].

We applied PIC2LNT on a benchmark of π-calculus specifications, which includes most of the examples provided with MWB (except those with self-recursion along the parallel operator), as well as unitary tests that we wrote ourselves. Our benchmark currently contains 160 files, which consist of about 2,000 lines of π-calculus and were translated in about 23,000 lines of LOTOS NT. This expansion in size is caused partly by the complexity of the translation (estimated at the end of Section 3) and partly by the fact that LOTOS NT is more verbose than the π-calculus (*e.g.*, LOTOS NT requires more keywords, gates have to be declared explicitly and passed as parameters to each process call, variables must be declared before usage, etc.).

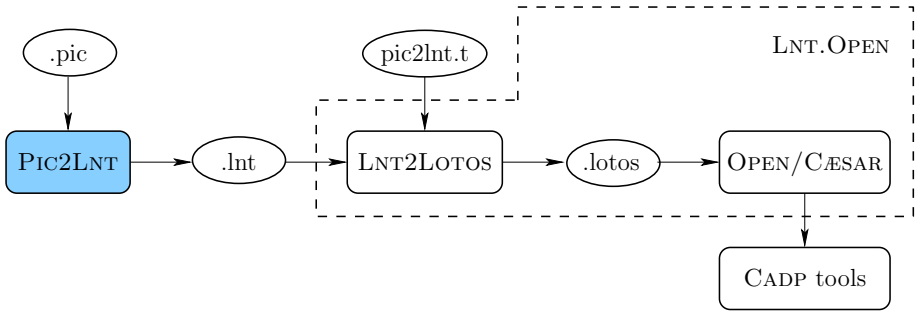

Fig. 3. Overview of the translation and verification process

Once a π-calculus specification is translated into LOTOS NT, the LNT.OPEN tool connects, by means of an intermediate translation into LOTOS (the pic2lnt.t file contains the external C definition of the function new_id()), the resulting specification to the OPEN/CÆSAR environment [10], which gives access to all the state-of-the-art on-the-fly verification tools of CADP. In particular, one can use the EVALUATOR 4.0 model checker [17] to verify temporal properties specified in MCL (an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators) involving channel names and/or data values present on transition labels. The counterexamples provided by the model checker are translated back into the π-calculus format by using the label renaming features provided by CADP.

5 Case Study: A Dispatcher Web Service

With the recent advent of Web services, the π-calculus has found a new application area. Many works have focused on the application of the π-calculus for modelling Web services and their composition, see *e.g.*, [7, 14, 25]. As far as implementation languages are concerned, BPEL is an XML-based executable language for implementing Web services orchestrations, and its specification [5] includes some dynamic communication primitives, namely endpoint references. As written in the BPEL specification: "*An endpoint reference makes it possible in* BPEL *to dynamically select a provider for a particular type of service and to invoke their operations*".

In this section, we present an example of Web service (a dispatcher) involving dynamicity in the system architecture. This service receives requests from some clients, and depending on the product searched by the client, forwards this request to the adequate server. The server receives this request from the dispatcher as well as a private channel (x), and uses this new channel to interact directly with the client. First, it sends to him/her some information about the product (*e.g.*, price, availability, etc.) and next it receives the client's final decision (*purchase* or *refuse*). The client stops as soon as (s)he accepts the purchase. We show

below the π-calculus specification of a system composed of one client and three servers selling different products (identified by a, b, c). In the specification, we use four private channels (*req*, a, b, and c) and three public channels (*request*, *purchase*, and *refuse*). Polyadic emissions are enclosed between angle brackets.

$$
\begin{aligned}
Main &= (\nu\ req, a, b, c)(Client(req, a, b, c) \mid Dispatcher(req) \mid \\
&\quad Server(a) \mid Server(b) \mid Server(c)) \\
Client(req, a, b, c) &= (\nu x)(\overline{request}\, a.\overline{req}\langle a, x\rangle.ClientAux(req, a, a, b, c, x)) + \\
&\quad (\nu x)(\overline{request}\, b.\overline{req}\langle b, x\rangle.ClientAux(req, b, a, b, c, x)) + \\
&\quad (\nu x)(\overline{request}\, c.\overline{req}\langle c, x\rangle.ClientAux(req, c, a, b, c, x)) \\
ClientAux(req, k, a, b, c, x) &= x(info).(\overline{x}\, purchase.\overline{purchase}\, k.0 + \\
&\quad \overline{x}\, refuse.\overline{refuse}\, k.Client(req, a, b, c)) \\
Dispatcher(req) &= req(k, x).\overline{k}\, x.Dispatcher(req) \\
Server(k) &= k(x).\overline{x}\, info.x(decision).Server(k)
\end{aligned}
$$

Figure 4(a) shows the LTS generated using the LNT.OPEN tool from the LOTOS NT specification (an excerpt of which can be found in Appendix A) produced by the PIC2LNT translator. The transition labels have been renamed using CADP to keep only the relevant information about channels. The system starts emitting a request for one of the three possible products (state 0). Then, the dispatcher interacts with the concerned server, and this server with the client. This corresponds to sequences of τ transitions in the LTS because private interactions result in hidden transitions according to the π-calculus semantics. At that point of the execution, if the client decides to purchase the product (states 13, 15, 17), the system terminates (state 19). If the client decides to refuse the purchase (states 14, 16, 18), state 20 is reached where the client can submit another request.

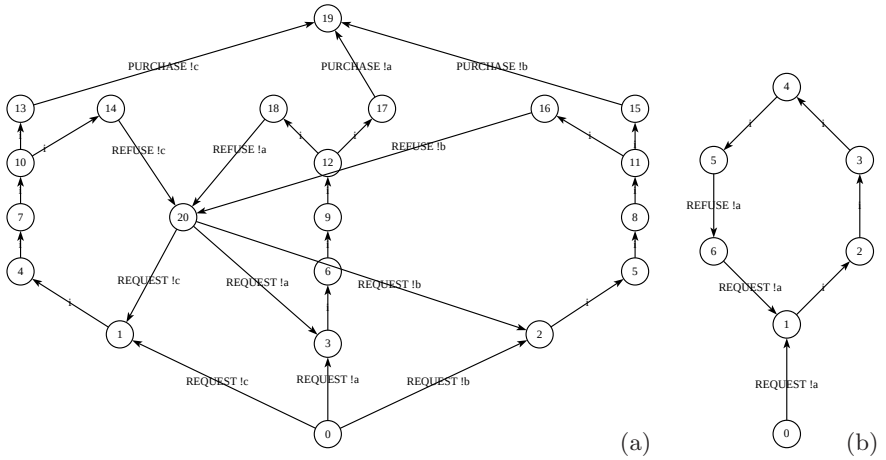

Fig. 4. (a) LTS of the dispatcher specification. (b) Lasso-shaped counterexample.

Next, we illustrate how the Evaluator 4.0 model checker [17] of Cadp can be used for analyzing this simple example. This tool accepts as input Mcl formulas expressing properties on actions but also on data parameters. In particular, for Lts models generated from π-calculus specifications, Mcl makes it possible to specify properties about channels appearing either as subject or object of a communication. For example, the Mcl formula below expresses that each request submitted by the client is eventually answered positively:

$$[true^*.\{request\ ?x{:}String\}]\ \mu X.(\langle true\rangle true \wedge [\neg\{purchase\ !x\}]X)$$

The first box modality matches all transition sequences starting at the initial state of the Lts and ending with a *request* action. The channel value (encoded as a character string) communicated via *request* is captured in the x variable, which is reused later on in the minimal fixed point formula expressing the inevitable reachability of a *purchase* action involving x. This formula fails on the Lts in Figure 4(a) because a client can indefinitely refuse a product, as illustrated in Figure 4(b) by the lasso-shaped counterexample exhibited by Evaluator 4.0.

The other Mcl formula below states that no positive or negative answer can be delivered for product a without a corresponding request being issued:

$$[(\neg\{request\ !"a"\})^*.\{purchase\ !"a"\} \vee \{refuse\ !"a"\}]false$$

This box modality forbids the existence of a sequence consisting of (0 or more) actions different from a *request* for a, followed by a *purchase* or a *refuse* action concerning a. Using Evaluator 4.0, we checked that this property holds on the Lts shown in Figure 4(a).

6 Related Work

During the past two decades, various approaches were followed for analyzing π-calculus specifications automatically. One of the first analysis tools specifically dedicated to the π-calculus was the Mobility Workbench (Mwb) [28], developed in the 90s for manipulating and analyzing mobile concurrent systems. The main features of Mwb are checking open bisimulation equivalences [26] and modal μ-calculus formulas using a sequent-calculus based model checker.

Other works considered automata-based representations of finite control π-calculus agents, with the goal of reusing the equivalence checkers and model checkers available for automata. Several decidability results about the strong and weak equivalence of π-calculus agents under certain assumptions on name spaces were presented in [6]. In a similar line of work, [21] introduce an *irredundant unfolding* notion that enables to check efficiently bisimilarity of finitary agents using an ordinary partition refinement algorithm, and also to minimize single agents. This line of work was continued in [8] by associating ordinary finite state automata to equivalent π-calculus agents using HD-automata as intermediate representation. This enabled to reuse the automata-based verification environment Jack for analyzing mobile processes, both by means of equivalence

checking (using the MAUTO tool) and by model checking formulas specified in π-logic (a variant of modal μ-calculus dedicated to the π-calculus), by translating them into ACTL and applying the AMC model checker of the JACK environment.

A logical encoding of the operational semantics of the π-calculus into MMC processes was proposed in [30]. MMC is a model checker for mobile systems which builds on XMC, a model checker for Milner's value passing CCS implemented using the XSB tabled logic-programming engine. This connection allows the specification of correctness properties in an expressive subset of the π-logic and their verification using MMC. A probabilistic / stochastic version of the π-calculus was considered in [23], where an automated procedure was proposed for generating first the corresponding symbolic transition graphs, and second Markov decision processes or continuous-time Markov chains. These models can be used as input of existing probabilistic model checkers such as PRISM, where properties are typically specified using the temporal logics PCTL and CSL.

Compared to these works, we chose to follow here a different approach, by translating a π-calculus specification into an equivalent description in a high-level language equipped with tools for the generation and manipulation of the underlying transition system. This translation-based approach was subject to several proposals concerning various languages. In [2], the authors show how to map the π-calculus into the MONSTR graph rewriting language. This work, which was not targeted to verification purposes, illustrated the convenience of representing an evolving network of communicating agents in a graph manipulation formalism, but also pointed out the heavy cost in practice of faithfully implementing the communication primitive of mobile process calculi.

Another way of analyzing π-calculus specifications, proposed independently in [29, 27], consists in translating them in PROMELA and verifying LTL formulas using the SPIN model checker. As regards channel mobility, PROMELA is suitable as a target language because it allows channel names to be communicated between processes as ordinary data values. The rule-based translation of [29] was implemented in the PI2PROMELA tool and successfully applied to model the Bluetooth service discovery protocol. In [27] a different translation is proposed (only for the monadic π-calculus) and implemented as an add-on to the MWB tool. The verification of LTL properties on the generated PROMELA code requires the manual specification of an environment whose role is to close the PROMELA description and to define the variables needed in the LTL formulas. Therefore, as observed in [27], the verification approach based on translation to PROMELA cannot be completely automated. Moreover, no attempt is made in [29, 27] to justify that the translations proposed preserve the π-calculus semantics. Indeed, PROMELA is not equipped with an LTS semantics and the underlying state space model is suited mainly for LTL model checking in the state-based setting, whereas the π-calculus semantics relies on LTSs and bisimulation relations.

Finally, another group of works aimed at modelling the mobility in the LOTOS specification language. A first method for modelling dynamic communication structures by encoding link names as data values was proposed in [9], together with a sufficient condition on the communication structures (binary group com-

munication) guaranteeing that the modelling is possible. This method is illustrated in [9] by specifying the handover procedure implemented in a mobile telecommunication network. In [22], the authors introduce M-LOTOS, a mobile extension of LOTOS based on the π-calculus, which preserves the other LOTOS specification styles. An operational semantics of M-LOTOS and a notion of early bisimulation are defined, and the usage of the language is illustrated by several examples, including an ODP trader.

Our translation from π-calculus to LOTOS NT makes the state-of-the-art verification tools of CADP directly available for analyzing π-calculus specifications: μ-calculus model checking with EVALUATOR [17], equivalence checking (branching, weak, etc.) with BISIMULATOR [3], compositional and distributed verification, rapid prototyping, etc.

7 Conclusion and Future Work

We have presented a translation from the finite control fragment of the π-calculus to LOTOS NT, which is one of the input languages of the CADP toolbox. Consequently, this translation makes it possible to use all the state-of-the-art verification tools of CADP to analyze π-calculus specifications. The translation is completely automated by the PIC2LNT tool and validated on many examples. The restriction to the finite control fragment of the π-calculus, first considered in [21, 6], does not hamper the practical usability of the language: even if the number of π-calculus agents must be statically known, the mobility of communication channels can be fully exploited.

We plan to continue our work by extending the π-calculus with data-handling features, with the goal of widening its possible application domains. This can be done by extending the language grammar and the translation to support typed variables and data expressions. As language for describing data, a natural candidate would be LOTOS NT itself: indeed, the data types and functions used in the π-calculus specification could be described in LOTOS NT and directly incorporated to the LOTOS NT code produced by translation. This would result in an applied π-calculus, such as the variant of the calculus proposed in [1] for the verification of security properties.

References

1. M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi-Calculus. *ACM Trans. Inf. Syst. Secur.*, 10(3), 2007.
2. R. Banach, J. Balazs, and G. Papadoupolous. A Translation of the Pi-Calculus Into MONSTR. *J. UCS*, 1(6):339–398, 1995.
3. D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu. BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking. In *Proc. of TACAS'05*, volume 3440 of LNCS, pages 581–585. Springer, 2005.
4. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.0). INRIA/VASY, 107 pages, Mar. 2010.

5. OASIS Technical Committee. Web Services Business Process Execution Language Version 2.0, 2007.
6. M. Dam. On the Decidability of Process Equivalences for the pi-Calculus. *Theor. Comput. Sci.*, 183(2):215–228, 1997.
7. S. Deng, Z. Wu, M. Zhou, Y. Li, and J. Wu. Modeling Service Compatibility with Pi-Calculus for Choreography. In *Proc. of ER'06*, volume 4215 of LNCS, pages 26–39. Springer, 2006.
8. G. L. Ferrari, G. Ferro, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. An Automated Based Verification Environment for Mobile Processes. In *Proc. of TACAS'97*, volume 1217 of LNCS, pages 275–289. Springer, 1997.
9. L.-Å. Fredlund and F. Orava. Modelling Dynamic Communication Structures in LOTOS. In *Proc. of FORTE'91*, volume C-2 of *IFIP Transactions*, pages 185–200. North-Holland, 1991.
10. H. Garavel. OPEN/CÆSAR: An Open Software Architecture for Verification, Simulation, and Testing. In *Proc. of TACAS'98*, volume 1384 of LNCS, pages 68–84. Springer, 1998.
11. H. Garavel, F. Lang, and R. Mateescu. Compiler Construction using LOTOS NT. In *Proc. of CC'02*, volume 2304 of LNCS, pages 9–13. Springer, 2002.
12. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of CAV'07*, volume 4590 of LNCS, pages 158–163. Springer, 2007.
13. H. Garavel and M. Sighireanu. A Graphical Parallel Composition Operator for Process Algebras. In *Proc. of FORTE/PSTV'99*, pages 185–202. IFIP, Kluwer Academic Publishers, Oct. 1999.
14. R. Lucchi and M. Mazzara. A Pi-Calculus based Semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.
15. ISO/IEC. Enhancements to LOTOS (E-LOTOS). International Standard 15437:2001, International Organization for Standardization, Genève, Sept. 2001.
16. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization, Genève, Sept. 1989.
17. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of LNCS, pages 148–164. Springer, 2008.
18. R. Milner. *Communication and Concurrency*, Prentice-Hall, 1989.
19. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
20. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.
21. U. Montanari and M. Pistore. Checking Bisimilarity for Finitary Pi-Calculus. In *Proc. of CONCUR'95*, volume 962 of LNCS, pages 42–56. Springer, 1995.
22. E. Najm, J.-B. Stefani, and A. Février. Towards a Mobile LOTOS. In *Proc. of FORTE'95*, volume 43 of *IFIP Conference Proceedings*, pages 127–142. Chapman & Hall, 1995.
23. G. Norman, C. Palamidessi, D. Parker, and P. Wu. Model Checking Probabilistic and Stochastic Extensions of the Pi-Calculus. *IEEE Trans. Software Eng.*, 35(2):209–223, 2009.
24. J. Parrow. An introduction to the pi-calculus, In *Handbook of Process Algebra*, chapter 8, pages 479–544. North-Holland, 2001.
25. F. Puhlmann. Why Do We Actually Need the Pi-Calculus for Business Process Management? In *Proc. of BIS'06*, volume 85 of *LNI*, pages 77–89. GI, 2006.

26. D. Sangiorgi. A Theory of Bisimulation for the pi-Calculus. *Acta Inf.*, 33(1):69–97, 1996.
27. H. Song and K. J. Compton. Verifying Pi-Calculus Processes by Promela Translation. Technical Report CSE-TR-472-03, University of Michigan, USA, 2003.
28. 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. Springer, 1994.
29. P. Wu. Interpreting Pi-Calculus with Spin/Promela. In *Proc. of NCTCS'03*, volume 8 (supplement) of *Computer Science*, pages 7–9, 2003.
30. P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A Logical Encoding of the Pi-Calculus: Model Checking Mobile Processes using Tabled Resolution. *STTT*, 6(1):38–66, 2004.

A Dispatcher Web Service Translated to LOTOS NT

We show below an excerpt of the LOTOS NT code (processes MAIN and Dispatcher_4 corresponding to the agents *Main* and *Dispatcher*) generated by the PIC2LNT translator from the π-calculus specification of the dispatcher Web service given in Section 5. The names of LOTOS NT processes are indexed by the number of gates in $\overline{G}$ (not counting the G_{pub} and G_{priv} gates).

```
process MAIN [PUBLIC,PRIVATE:any] is
  var req, a, b, c:Chan in
    req:=req(new_id()); a:=a(new_id()); b:=b(new_id()); c:=c(new_id());

    hide G0:any in par G0 in hide G1:any in par G1 in
    hide G2:any in par G2 in hide G3:any in par G3 in
       Client_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,a,b,c,2)
    || Dispatcher_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,6) end par end hide
    || Server_3 [PUBLIC,PRIVATE,G0,G1,G2] (a,14)         end par end hide
    || Server_2 [PUBLIC,PRIVATE,G0,G1] (b,30)            end par end hide
    || Server_1 [PUBLIC,PRIVATE,G0] (c,31)               end par end hide
  end var
end process

process Dispatcher_4[PUBLIC,PRIVATE,G0,G1,G2,G3:any](req:Chan,pid:Nat) is
  select var k,x:Chan, s:Nat in
    G0 (!req, ?k, ?x, ?s, !pid) [] G1 (!req, ?k, ?x, ?s, !pid) []
    G2 (!req, ?k, ?x, ?s, !pid) [] G3 (!req, ?k, ?x, ?s, !pid) []
    PUBLIC (!req, ?k, ?x, !false) where is_public(req) []
    PRIVATE (!req, ?k, ?x, !false) where not(is_public(req))
  end select ;
  select var r:Nat in
    G0 (!k, !x, !pid, ?r) [] G1 (!k, !x, !pid, ?r) []
    G2 (!k, !x, !pid, ?r) [] G3 (!k, !x, !pid, ?r) []
    PUBLIC (!k, !x, !true) where is_public(k) []
    PRIVATE (!k, !x, !true) where not(is_public(k))
  end select ; Dispatcher_4 [PUBLIC,PRIVATE,G0,G1,G2,G3] (req,pid)
end process
```