Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 194 (2007) 39–60

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

State-oriented Noninterference for CCS

Ilaria Castellani[1,2]

INRIA Sophia Antipolis
2004 route des Lucioles, BP 93,
06902 Sophia Antipolis Cedex, France

Abstract

We address the question of typing *noninterference* (NI) in the calculus CCS, in such a way that Milner's translation into CCS of a standard parallel imperative language preserves both an existing NI property and the associated type system. Recently, Focardi, Rossi and Sabelfeld have shown that a variant of Milner's translation, restricted to the sequential fragment of the language, maps a time-sensitive NI property to that of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC) on CCS. However, since CCS was not equipped with a type system, the question of whether the translation preserves types could not be addressed. We extend Focardi, Rossi and Sabelfeld's result by showing that a slightly simpler variant of Milner's translation preserves a *time-insensitive* NI property on the full parallel language, by mapping it again to PBNDC. As a by-product, we formalise a folklore result, namely that Milner's translation preserves a behavioural equivalence on programs. We present a simple type system ensuring PBNDC on CCS, inspired by existing type systems for the π-calculus. Unfortunately, this type system as it stands is too restrictive to grant the expected type preservation result. We sketch a solution to overcome this problem.

Keywords: Noninterference, type systems, parallel imperative languages, process calculi, bisimulation.

1 Introduction

The issue of *secure information flow* has attracted a great deal of interest in recent years, spurred by the spreading of mobile devices and nomadic computation. The question has been studied in some depth both for programming languages (see [26] for a review) and for process calculi [24,8,13,21,11,14,12,5,17,10]. We shall speak of "language-based security" when referring to programming languages, and of "process-based security" when referring to process calculi.

The language-based approach is concerned with secret *data* not being leaked by programs, that is, with the security property of confidentiality. This property is usually formalized via the notion of *noninterference* (NI), stating that secret inputs of programs should not influence their public outputs, since this could allow - at least in principle - a public user to reconstruct secret information.

[1] Work partially supported by the ANR SETI-06-010 grant.

[2] Email: Ilaria.Castellani@sophia.inria.fr

doi:10.1016/j.entcs.2007.09.012

The process-based approach, on the other hand, is concerned with secret *actions* of processes not being publicly observable. Although bearing a clear analogy with the language-based approach - security levels are assigned in both cases to information carriers, respectively variables and channels - the process-based approach does not rely on quite the same simple intuition. Indeed, there are several choices as to what an observer can gather by communicating with a process. This is reflected in the variety of NI properties proposed for process calculi, mostly based on trace equivalence, testing or bisimulation (cf [8] for a review). In general, these properties do not clearly distinguish between the flow of data and the flow of control, which are closely intertwined in process calculi. Let us consider some examples.

In the calculus CCS with value passing [18], an input process $a(x).\, P$ receives a value v on channel a and then becomes $P\{v/x\}$. Symmetrically, an output process $\overline{a}\langle e\rangle.\, P$ emits the value of expression e on channel a and then becomes P. Then a typical insecure data flow is the following, where subscripts indicate the security level of channels (h meaning "high" or "secret", and ℓ meaning "low" or "public"):

$$get_h(x).\, \overline{put}_\ell\langle x\rangle$$

Here a value received on a high channel is retransmitted on a low channel. Since the value for x may be obtained from some high external source, this process is considered insecure. However, there are other cases where low output actions carry no data, or carry data that do not originate from a high source, as in:

$$get_h(x).\, \overline{a}_\ell \qquad\qquad c_h.\, \overline{put}_\ell\langle v\rangle \qquad\qquad get_h(x).\, \overline{put}_\ell\langle v\rangle$$

where a_ℓ, c_h are channels without parameters and v is a constant value. Although these processes do not directly transfer data from high to low level, they can be used to implement indirect insecure flows, as in the following process (where x is a boolean and actions on channels c_h and d_h are restricted and thus not observable):

$$P = ((get_h(x).\, \texttt{if } x \texttt{ then } \overline{c}_h \texttt{ else } \overline{d}_h) \mid (c_h.\, \overline{put}_\ell\langle 0\rangle \,+\, d_h.\, \overline{put}_\ell\langle 1\rangle)) \setminus \{c_h, d_h\}$$

The above examples suggest a simple criterion for enforcing noninterference on CCS, namely that high actions should not be followed by low actions. Admittedly, this requirement is very strong. However, it may serve, and indeed has been used, as a basis for defining *security type systems* for process calculi.

In the language-based approach, theoretical results have often lead to the design of tools for verifying security properties and to the development of secure implementations. Most of the languages examined so far have been equipped with a type system or some other tool to enforce the desired security property [19,20,23,22].

By contrast, the process-based approach has remained at a more theoretical level. Type systems for variants of the π-calculus, which combine the control of security with other correctness concerns, have been proposed by Hennessy et al. in [11,12] and by Honda et al. in [13,14]. A purely security type system for the π-calculus was presented by Pottier in [21]. More recently, different security type systems for the π-calculus were studied by Crafa and Rossi [10] and by Kobayashi [17] (this last work also provides a type inference algorithm). Other static verification methods have been proposed for a variant of CCS in [5].

We address the question of unifying the language-based and process-based approaches, by relating both their security notions and the associated type systems. A first step in this direction was taken by Honda, Vasconcelos and Yoshida in [13], where a parallel imperative language was embedded into a typed π-calculus. This work was pursued by Honda and Yoshida in [14], where more powerful languages, both imperative and functional, were considered. In [9], Focardi, Rossi and Sabelfeld showed that a variant of Milner's translation of a sequential imperative language into CCS preserves a *time-sensitive* NI property, by mapping it to the property of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), introduced by Focardi and Rossi in [7]. However, since CCS was not equipped with a security type system, the question of type preservation could not be addressed.

Taking [9] as our starting point, we extend its result by showing that a simpler variant of Milner's translation preserves a *time-insensitive* NI property on a parallel imperative language, by mapping it again to PBNDC. As a by-product, we show that the translation preserves a behavioural equivalence on programs. We also propose a type system for ensuring PBNDC, inspired by the type systems of [21,11,12] for the π-calculus. Unfortunately, this type system is too restrictive as it stands to reflect any of the known type systems for the source language. However, it can be used as a basis to derive a suitable type system, which is briefly sketched here.

The rest of the paper is organised as follows. In Section 2 we recall the definitions of BNDC and PBNDC for CCS and we present a type system ensuring the latter. In Section 3 we introduce the parallel imperative language and the time-insensitive NI property for it; we then propose an adaptation of Milner's translation of this language into CCS, and show that it preserves our NI property. We conclude with a discussion about type preservation. Proofs are omitted and may be found in [6].

2 A simple security type system for CCS

In this section we present a security type system for CCS, inspired by those proposed for the π-calculus by Pottier [21] and by Hennessy and Riely [11,12]. We prove that this type system ensures the property of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), introduced by Focardi and Rossi in [7].

2.1 The process calculus CCS

Our chosen process calculus is CCS with value passing and guarded sums. We start by recalling the main definitions. We assume a countable set of channels or names $\mathcal{N}$, ranged over by a, b, c, with the usual notational conventions for input and output. Similarly, let Var be a countable set of variables, disjoint from $\mathcal{N}$ and ranged over by x, y, z, and Val be the set of data values, ranged over by v, v'. We define Exp, ranged over by e, e', to be the set of boolean and arithmetic expressions built from values and variables using the standard total operations. Finally, we let $val : Exp \rightarrow Val$ be the evaluation function for expressions, satisfying $val(v) = v$ for any value v. We will use the notation $\vec{x}$ (resp. $\vec{v}$ or $\vec{e}$) to denote a sequence $\langle x_1, \ldots, x_n \rangle$ (resp. a sequence $\langle v_1, \ldots, v_n \rangle$ or $\langle e_1, \ldots, e_n \rangle$).

The syntax of process *prefixes*, ranged over by π, π', is given by:

$$\pi ::= a(x) \mid \overline{a}\langle e\rangle \mid a \mid \overline{a}$$

Simple prefixes of the form a and $\overline{a}$ will be used in examples but omitted from our technical treatment, since they are a simpler case of $a(x)$ and $\overline{a}\langle e\rangle$.

To define recursive processes, we assume a countable set $\mathcal{I} = \{A, B, \ldots\}$ of parametric process identifiers, each of which is supposed to have a fixed arity. We then define the set of *parametric terms*, ranged over by T, T', as follows:

$$T ::= A \mid (\texttt{rec}\, A(\vec{x})\,.\,P)$$

where P is a CCS process, as defined next.

A term $(\texttt{rec}\, A(\vec{x})\,.\,P)$ is supposed to satisfy some standard requirements: (1) all variables in $\vec{x}$ are distinct; (2) the length of $\vec{x}$ is equal to the arity of A; (3) all free variables of P belong to $\vec{x}$; (4) no free process identifier other than A occurs in P; (5) recursion is guarded: all occurrences of A in P appear under a prefix.

The set $\mathcal{P}r$ of *processes*, ranged over by P, Q, R, is given by the syntax:

$$P, Q ::= \sum_{i\in I} \pi_i.P_i \mid (P \mid Q) \mid (\nu a)\, P \mid T(\vec{e})$$

where I is an indexing set. We use $\mathbf{0}$ as an abbreviation for the empty sum $\sum_{i\in\emptyset} \pi_i.P_i$. Also, we abbreviate a unary sum $\sum_{i\in\{1\}} \pi_i.P_i$ to $\pi_1.P_1$ and a binary sum $\sum_{i\in\{1,2\}} \pi_i.P_i$ to $(\pi_1.P_1 + \pi_2.P_2)$. In a process $A(\vec{e})$ or $(\texttt{rec}\, A(\vec{x})\,.\,P)(\vec{e})$, the length of $\vec{e}$ is assumed to be equal to the arity of A. Finally, if $\vec{a} = \langle a_1, \ldots, a_n\rangle$, with $a_i \neq a_j$ for $i \neq j$, the term $(\nu a_1)\cdots(\nu a_n)\, P$ is abbreviated to $(\nu\vec{a})\, P$. If $K = \{a_1, \ldots, a_n\}$, we sometimes render $(\nu\vec{a})\, P$ simply as $(\nu K)\, P$, or use the original CCS notation $P\backslash K$, especially in examples.

The set of free variables (resp. free process identifiers) of process P will be denoted by $fv(P)$ (resp. $fid(P)$). We use $P\{v/x\}$ for the substitution of the variable x by the value v in P. Also, if $\vec{x} = \langle x_1, \ldots, x_n\rangle$ and $\vec{v} = \langle v_1, \ldots, v_n\rangle$, we denote by $P\{\vec{v}/\vec{x}\}$ the substitution of each variable x_i by the value v_i in P. Finally, $P\{T/A\}$ stands for the substitution of the parametric term T for the identifier A in P.

The semantics of processes is given by labelled transitions of the form $P \xrightarrow{\alpha} P'$. Transitions are labelled by *actions* α, β, γ, which are elements of the set:

$$Act \overset{\text{def}}{=} \{av : a \in \mathcal{N},\, v \in Val\} \cup \{\bar{a}v : a \in \mathcal{N},\, v \in Val\} \cup \{\tau\}$$

The subject of a prefix is defined by $subj(a(x)) = subj(\overline{a}\langle e\rangle) = a$, and the subject of an action by $subj(av) = subj(\bar{a}v) = a$ and $subj(\tau) = \tau$. The complementation operation is extended to input and output actions by letting $\overline{av} = \bar{a}v$ and $\overline{\bar{a}v} = av$.

The operational rules for CCS processes are recalled in Figure 1. A nondeterministic sum $\sum_{i\in I} \pi_i.P_i$ executes one summand $\pi_i.P_i$, simultaneously discarding the others. A summand $a(x).\, P_i$ receives a value v on channel a and then replaces it for x in P_i. A summand $\overline{a}\langle e\rangle.\, P_i$ emits the value of expression e on channel a and then becomes P_i. The parallel composition $P \mid Q$ interleaves the executions of P and Q, possibly synchronising them on complementary actions to yield a τ-action. The restriction $(\nu b)P$ behaves like P where actions on channel b are forbidden.

$$\text{(SUM-OP}_1\text{)} \quad \textstyle\sum_{i\in I} \pi_i.P_i \xrightarrow{av} P_i\{v/x\}, \ \text{ if } \ \pi_i = a(x) \ \text{ and } \ v \in Val$$

$$\text{(SUM-OP}_2\text{)} \quad \textstyle\sum_{i\in I} \pi_i.P_i \xrightarrow{\overline{a}v} P_i\,, \ \text{ if } \ \pi_i = \overline{a}\langle e\rangle \ \text{ and } \ val(e) = v$$

$$\text{(PAR-OP}_1\text{)} \quad \frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \qquad \text{(PAR-OP}_2\text{)} \quad \frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'}$$

$$\text{(PAR-OP}_3\text{)} \quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{(RES-OP)} \quad \frac{P \xrightarrow{\alpha} P' \quad b \neq subj(\alpha)}{(\nu b)P \xrightarrow{\alpha} (\nu b)P'}$$

$$\text{(REC-OP)} \quad \frac{P\{\vec{v}/\vec{x}\}\{\,(\texttt{rec}\,A(\vec{x})\,.\,P)\,/\,A\,\} \xrightarrow{\alpha} P' \qquad \vec{v} = val(\vec{e})}{(\texttt{rec}\,A(\vec{x})\,.\,P)(\vec{e}) \xrightarrow{\alpha} P'}$$

Fig. 1. Operational Semantics of CCS Processes

2.2 Security properties for CCS

We review two security properties for CCS: *Bisimulation-based Non Deducibility on Compositions* (BNDC), introduced by Focardi and Gorrieri [8] and reformulated by Focardi and Rossi in [7], and *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), proposed in [7] as a strenghtening of BNDC, better suited to deal with dynamic contexts.

We start by recalling the definition of weak bisimulation. We adopt the usual notational conventions:

- For any $\alpha \in Act$, let $P \overset{\alpha}{\Longrightarrow} P' \ \overset{\text{def}}{=} \ P \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} P'$
- For any $\alpha \in Act$, let $P \overset{\hat{\alpha}}{\Longrightarrow} P' \ \overset{\text{def}}{=} \ \begin{cases} P \overset{\alpha}{\Longrightarrow} P' & \text{if } \ \alpha \neq \tau \\ P \xrightarrow{\tau}{}^{*} P' & \text{if } \ \alpha = \tau \end{cases}$

Definition 2.1 [Weak Bisimulation] A symmetric relation $\mathcal{S} \subseteq (\mathcal{P}r \times \mathcal{P}r)$ is a *weak bisimulation* if $P \ \mathcal{S} \ Q$ implies, for any $\alpha \in Act$:

$$\text{If } P \xrightarrow{\alpha} P' \text{ then there exists } Q' \text{ such that } Q \overset{\hat{\alpha}}{\Longrightarrow} Q' \text{ and } P' \ \mathcal{S} \ Q'.$$

P and Q are *weakly bisimilar*, $P \approx Q$, if $P \ \mathcal{S} \ Q$ for some weak bisimulation $\mathcal{S}$.

To set up the scenario for BNDC, we need a few more definitions.

Definition 2.2 [High and low channels] The set $\mathcal{N}$ of channels is partitioned into a subset of high (secret) channels $\mathcal{H}$ and a subset of low (public) channels $\mathcal{L}$.

Input and output actions are then defined to be high or low according to the level of their supporting channel. No security level is given to τ-actions.

Definition 2.3 [Syntactically high processes w.r.t. $\mathcal{H}$]
The set of syntactically high processes with respect to $\mathcal{H}$, denoted $\mathcal{P}r^{\mathcal{H}}_{\mathsf{syn}}$, is the set of processes that contain only channels in $\mathcal{H}$.

The property of *Bisimulation-based Non Deducibility on Compositions* (BNDC) of [8], in its reformulation given by Focardi and Rossi [7], is now defined as follows:

Definition 2.4 [$\mathsf{BNDC}_{\mathcal{H}}$] Let $P \in \mathcal{P}r$ and $\mathcal{H} \subseteq \mathcal{N}$ be the set of high channels. Then P is *secure* with respect to $\mathcal{H}$, $P \in \mathsf{BNDC}_{\mathcal{H}}$, if for every process $\Pi \in \mathcal{P}r^{\mathcal{H}}_{\mathsf{syn}}$, $(\nu\mathcal{H})(P \mid \Pi) \approx (\nu\mathcal{H})P$.

When there is no ambiguity, we write simply BNDC instead of $\mathsf{BNDC}_{\mathcal{H}}$. Let us point out two typical sources of insecurity:

(i) Insecurity may appear when a high name is followed by a low name in P, because in this case the execution of $(\nu\mathcal{H})P$ may block on the high name, making the low name unreachable, while it is always possible to find a high process Π that makes the low name reachable in $(\nu\mathcal{H})(P \mid \Pi)$. Let for instance $P = a_h.\,\overline{b_\ell}$. Choosing $\Pi = \overline{a_h}$, one obtains $(\nu\mathcal{H})(P \mid \Pi) \not\approx (\nu\mathcal{H})P$.

(ii) Insecurity may also appear when a high name is in conflict with a low name, as in $P = a_h + \overline{b_\ell}$. Here again, taking $\Pi = \overline{a_h}$ one gets $(\nu\mathcal{H})(P \mid \Pi) \not\approx (\nu\mathcal{H})P$, since the first process can do a silent move $\xrightarrow{\tau}$ leading to a state equivalent to $\mathbf{0}$, which the second process cannot match. Note on the other hand that $Q = a_h.\,\overline{b_\ell} + \overline{b_\ell}$ is secure, because in this case, the synchronisation on channel a_h in $(\nu\mathcal{H})(Q \mid \Pi)$ may be simulated by inaction in $(\nu\mathcal{H})Q$.

In [7], Focardi and Rossi proposed a more robust property than BNDC, which they called *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC).

To define PBNDC, a more permissive notion of bisimulation is required, based on a new transition relation $\overset{\tilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$, defined for any $\alpha \in Act$ by:

$$P \overset{\tilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} P' \quad \overset{\text{def}}{=} \quad \begin{cases} P \overset{\hat{\alpha}}{\Longrightarrow} P' \text{ or } P \xrightarrow{\tau}{}^{*} P' & \text{if } \ subj(\alpha) \in \mathcal{H} \\ P \overset{\hat{\alpha}}{\Longrightarrow} P' & \text{otherwise} \end{cases}$$

Definition 2.5 [Weak bisimulation up-to-high]
A symmetric relation $\mathcal{S} \subseteq (\mathcal{P}r \times \mathcal{P}r)$ is a *weak bisimulation up to high* if $P \;\mathcal{S}\; Q$ implies, for any $\alpha \in Act$:

$$\text{If } P \xrightarrow{\alpha} P' \text{ then there exists } Q' \text{ such that } Q \overset{\tilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} Q' \text{ and } P' \;\mathcal{S}\; Q'.$$

Two processes P, Q are *weakly bisimilar up to high*, written $P \approx_{\mathcal{H}} Q$, if $P \;\mathcal{S}\; Q$ for some weak bisimulation up to high $\mathcal{S}$.

Definition 2.6 [$\mathsf{PBNDC}_{\mathcal{H}}$] Let $P \in \mathcal{P}r$. Then P is said to be *persistently secure* with respect to $\mathcal{H}$, $P \in \mathsf{PBNDC}_{\mathcal{H}}$, if $P \approx_{\mathcal{H}} (\nu\mathcal{H})P$.

The transition relation $\overset{\tilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$ is used in the definition of PBNDC to allow high moves of P to be matched by (possibly empty) sequences of τ-moves of $(\nu\mathcal{H})P$.

It was shown in [7] that PBNDC is stronger than BNDC, and that requiring PBNDC for P amounts to requiring BNDC for all reachable states of P. All the examples considered above are treated in the same way by BNDC and PBNDC. Examples of secure but not persistently secure processes may be found in [7,6].

2.3 *A security type system for* PBNDC

In this section we present our security type system for CCS and we show that it ensures the PBNDC property. This type system is inspired by those previously proposed for the π-calculus by Pottier [21] and by Hennessy et al. [11,12].

Security levels, ranged over by δ, θ, σ, are defined as usual to form a lattice $(\mathcal{T}, \leq)$, where the order relation $\leq$ stands for "less secret than". Here we assume the lattice to be simply $\{\ell, h\}$, with $\ell \leq h$, to match the partition of the set of channels into $\mathcal{L}$ and $\mathcal{H}$.

A *type environment* Γ is a mapping from channels to security levels, together with a partial mapping from process identifiers to security levels. This mapping is extended to prefixes and visible actions by letting $\Gamma(\pi) = \Gamma(\mathit{subj}(\pi))$ and for any $\alpha \neq \tau$, $\Gamma(\alpha) = \Gamma(\mathit{subj}(\alpha))$. Type judgements for processes have the form $\Gamma \vdash_\sigma P$. Intuitively, $\Gamma \vdash_\sigma P$ means that in the type environment Γ, σ is a *lower bound* on the security level of channels occurring in P. The typing rules are as follows:

(Sum)
$$\frac{\forall i \in I: \quad \Gamma(\pi_i) = \sigma \qquad \Gamma \vdash_\sigma P_i}{\Gamma \vdash_\sigma \sum_{i\in I} \pi_i.P_i}$$

(Par)
$$\frac{\Gamma \vdash_\sigma P \qquad \Gamma \vdash_\sigma Q}{\Gamma \vdash_\sigma P \mid Q}$$

(Res)
$$\frac{\Gamma, b : \theta \vdash_\sigma P}{\Gamma \vdash_\sigma (\nu b)P}$$

(Sub)
$$\frac{\Gamma \vdash_\sigma P \qquad \sigma' \leq \sigma}{\Gamma \vdash_{\sigma'} P}$$

(Rec$_1$)
$$\frac{\Gamma(A) = \sigma}{\Gamma \vdash_\sigma A(\vec{e})}$$

(Rec$_2$)
$$\frac{\Gamma, A : \sigma \vdash_\sigma P}{\Gamma \vdash_\sigma (\mathtt{rec}\ A(\vec{x}) \,.\, P)(\vec{e})}$$

Let us briefly discuss rule (Sum), which is the less standard one. This rule imposes a strong constraint on processes $\sum_{i\in I} \pi_i.P_i$, namely that all prefixes π_i have the same security level σ and that the P_i have themselves type σ. In fact, since each judgement $\Gamma \vdash_\sigma P_i$ may have been derived using subtyping, this means that originally $\Gamma \vdash_{\sigma_i} P_i$ for some σ_i such that $\sigma \leq \sigma_i$. Note that, as expected, $a_h.\,\overline{b_\ell}$ and $a_h + \overline{b_\ell}$ are not typable. However, it should be pointed out that the secure process $a_h.\,\overline{b_\ell} + \overline{b_\ell}$ is not typable either. Hence, rule (Sum) is stricter than we would wish. In order to make process $a_h.\,\overline{b_\ell} + \overline{b_\ell}$ typable, we could envisage replacing rule (Sum) by the following rule (SUM-LAX), which allows a prefix π_i to be of level higher than ℓ, provided its continuation process P_i is indistinguishable from the original sum process:

$$(\text{SUM-LAX}) \quad \frac{\forall i \in I: \quad \Gamma \vdash_\sigma P_i \wedge (\Gamma(\pi_i) = \sigma \vee (\Gamma(\pi_i) > \sigma \wedge P_i \approx_{\mathcal{H}} \sum_{i\in I} \pi_i.P_i))}{\Gamma \vdash_\sigma \sum_{i\in I} \pi_i.P_i}$$

Note however that rule (SUM-LAX) makes use of the semantic equivalence $\approx_{\mathcal{H}}$, and thus is not completely static. To avoid introducing such a semantic check in our type system, we shall stick, for the time being, to the more classical rule (SUM).

We proceed now to establish the soundness of this type system for PBNDC. We state here the most relevant results, referring the reader to [6] for more details.

Theorem 2.7 (Subject reduction)
For any $P \in \mathcal{P}r$, if $\Gamma \vdash_\sigma P$ and $P \xrightarrow{\alpha} P'$ then $\Gamma \vdash_\sigma P'$.

Lemma 2.8 (Confinement)
Let $P \in \mathcal{P}r$ and $\Gamma \vdash_\sigma P$. If $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then $\Gamma(\alpha) \geq \sigma$.

The key for the soundness proof is the following property of typable programs:

Lemma 2.9 ($\approx_{\mathcal{H}}$– invariance under high actions)

Let $P \in \mathcal{P}r$, $\Gamma \vdash_\sigma P$ and $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$. If $P \xrightarrow{\alpha} P'$ and $\Gamma(\alpha) = h$ then $P \approx_{\mathcal{H}} P'$.

Corollary 2.10 (Compositionality of $\approx_{\mathcal{H}}$ for typable programs)
Let $P, Q, R \in \mathcal{P}r$ and Γ be a type environment such that $\Gamma \vdash_\sigma P$, $\Gamma \vdash_\sigma Q$ and $\Gamma \vdash_\sigma R$. Then, if $P \approx_{\mathcal{H}} Q$, also $P \mid R \approx_{\mathcal{H}} Q \mid R$.

Note that $\approx_{\mathcal{H}}$ is not preserved by parallel composition on arbitrary programs, as shown by this example, where $P_i \approx_{\mathcal{H}} Q_i$ for $i = 1, 2$ but $P_1 \mid P_2 \not\approx_{\mathcal{H}} Q_1 \mid Q_2$:

$$P_1 = a_h \qquad Q_1 = \mathbf{0} \qquad P_2 = Q_2 = b_\ell + \overline{a_h}$$

It is easy to see that $P_1 \mid P_2 \not\approx_{\mathcal{H}} Q_1 \mid Q_2$, since $P_1 \mid P_2$ can perform a τ-action which $Q_1 \mid Q_2$ cannot match. Note that P_2 is not typable. In fact P_2 is insecure. Indeed, the property of PBNDC itself is compositional, as shown in [7].

Using the above results, we may show that typability implies PBNDC:

Theorem 2.11 (Soundness)
If $P \in \mathcal{P}r$ and $\Gamma \vdash_\sigma P$ then $P \approx_{\mathcal{H}} (\nu\mathcal{H})P$, where $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$.

This concludes, for the time being, our discussion about security and types for CCS.

3 Translating parallel imperative programs into CCS

We focus here on the parallel imperative language studied by Smith and Volpano in [30], which we call PARIMP. Several NI properties and related type systems have been already proposed for this language [27,1,29,4], inspired by the pioneering work of Volpano, Smith et al. [32,30,31]. There exists a well known translation of PARIMP into CCS, presented by Milner in [18]. In [9], Focardi, Rossi and Sabelfeld showed that a variant of this translation preserves – by mapping it to PBNDC – a time-sensitive notion of NI for the sequential fragment of PARIMP. We shall be concerned here with the full language PARIMP, and with a *time-insensitive* NI property for this language. We will prove that this NI property is preserved by a suitable variant of Milner's translation. As a by-product, we will show that our translation also preserves a behavioural equivalence on programs.

3.1 The imperative language PARIMP

In this section we recall the syntax and semantics of the language PARIMP, and we define a time-insensitive NI property for it, inspired by that of [4].

We assume a countable set of variables ranged over by X, Y, Z, a set of values ranged over by V, V', and a set of expressions ranged over by E, E'. Formally, *expressions* are built using total functions $F, G, \ldots$, which we assume to be in a 1 to 1 correspondence with the functions $f, g, \ldots$ used to build CCS expressions:

$$E ::= F(X_1, \ldots, X_n)$$

The set $\mathcal{C}$ of *programs* or *commands*, ranged over by C, D, is defined by:

$$C,\, D ::= \mathtt{nil} \;\mid\; X := E \;\mid\; C\,;D \;\mid\; (\mathtt{if}\; E\; \mathtt{then}\; C\; \mathtt{else}\; D) \;\mid$$
$$(\mathtt{while}\; E\; \mathtt{do}\; C) \;\mid\; (C \parallel D)$$

The operational semantics of the language is given in terms of transitions between configurations $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ where C, C' are programs and s, s' are *states* or *memories*, that is, mappings from a finite subset of variables to values. These mappings are extended in the obvious way to expressions, whose evaluation is assumed to be terminating and atomic. We use the notation $s[V/X]$ for memory update, $\mapsto$ for the reflexive closure of $\rightarrow$, and $\rightarrow^*$ for its reflexive and transitive closure. The operational rules for configurations are given in Figure 2. The rules (ParL-Op2) and (ParR-Op2) are introduced, as in [3], to allow every terminated configuration to take the form $\langle \mathtt{nil}, s\rangle$.

A configuration $\langle C, s\rangle$ is *well-formed* if $fv(C) \subseteq \mathtt{dom}(s)$. It is easy to see, by inspection of the rules, that $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ implies $fv(C') \subseteq fv(C)$ and $\mathtt{dom}(s') = \mathtt{dom}(s)$. Hence well-formedness is preserved by execution.

As for CCS, we assume variables to be partitioned into a set of *low variables* L and a set of *high variables* H. In examples, we will use the subscripts L and H for variables belonging to the sets L and H, respectively. We may now introduce the notions of low-equality and low-bisimulation.

Definition 3.1 [L-Equality] Two memories s and t are L-*equal*, written $s =_L t$, if $\mathtt{dom}(s) = \mathtt{dom}(t)$ and $(X \in \mathtt{dom}(s) \cap L \;\Rightarrow\; s(X) = t(X))$.

Definition 3.2 [L-Bisimulation]

A symmetric relation $\mathcal{S} \subseteq (\mathcal{C} \times \mathcal{C})$ is a L-*bisimulation* if $C\,\mathcal{S}\,D$ implies, for any pair of states s and t such that $s =_L t$ and $\langle C, s\rangle$ and $\langle D, t\rangle$ are well-formed:

$$\text{If } \langle C, s\rangle \to \langle C', s'\rangle, \text{ then there exist } D', t' \text{ such that}$$
$$\langle D, t\rangle \mapsto \langle D', t'\rangle \text{ and } s' =_L t' \text{ and } C' \,\mathcal{S}\, D'.$$

Two programs C, D are L-*bisimilar*, $C \simeq_L D$, if $C \,\mathcal{S}\, D$ for some L-bisimulation $\mathcal{S}$.

Note that the simulating program is required to mimic each move of the first program by either one or zero moves. This notion of low-bisimulation is inspired from [4]. We could have chosen a weaker notion, where $\mapsto$ is replaced by $\to^*$, as proposed in [27]. However our choice allows for a more precise notion of security, which respects "state-traces", as illustrated by Example 3.4 below.

Definition 3.3 [L-Security] A program C is L-*secure* if $C \simeq_L C$.

When L is clear, we shall speak simply of *low-equality*, *low-bisimulation* and *security*.

Example 3.4 The following program, where $\texttt{loop}\ D \overset{\text{def}}{=} (\texttt{while}\ tt\ \texttt{do}\ D)$:

$$C = (\texttt{if}\ X_H = 0\ \texttt{then}\ \texttt{loop}\ (Y_L := 0\,;\, Y_L := 1)\ \texttt{else}\ \texttt{loop}\ (Y_L := 1\,;\, Y_L := 0))$$

is not L-secure since the branches of the conditional cannot simulate each other's moves in one or zero steps. However it would be secure according to the weaker notion of L-bisimulation obtained by replacing $\mapsto$ with $\to^*$ in Definition 3.2.

3.2 Milner's translation of PARIMP into CCS

We now review Milner's translation of the language PARIMP into CCS [18]. This translation makes use of two new constructs of CCS, renaming and conditional, whose semantics we assume to be known (see [18], or the full paper [6]).

First, *registers* are introduced to model the store. For each program variable X, the associated register Reg_X, parameterised by the value it contains, is defined by:

$$Reg_X(v) \overset{\text{def}}{=} put_X(x).Reg_X(x) + \overline{get_X}\langle v\rangle.Reg_X(v)$$

The translation $[\![s]\!]$ of a state s is then a *pool of registers*, given by :

$$[\![s]\!] = Reg_{X_1}(s(X_1)) \mid \cdots \mid Reg_{X_n}(s(X_n)) \qquad \text{if } \texttt{dom}(s) = \{X_1, \ldots, X_n\}$$

The translation $[\![E]\!]$ of an expression $E = F(X_1, \ldots, X_n)$ is a process which fetches the values of registers $Reg_{X_1} \ldots, Reg_{X_n}$ into the variables $x_1, \ldots, x_n$ and then transmits over a special channel `res` the result of evaluating $f(x_1, \ldots, x_n)$, where f is the CCS function corresponding to the PARIMP function F:

$$[\![F(X_1, \ldots, X_n)]\!] = get_{X_1}(x_1). \cdots .get_{X_n}(x_n). \overline{\texttt{res}}\langle f(x_1, \ldots, x_n)\rangle. \mathbf{0}$$

The channel `res` is used by the auxiliary operator *Into*, defined by:

$$P\ Into\,(x)\ Q \overset{\text{def}}{=} (P \mid \texttt{res(x)}.\,Q)\backslash\texttt{res}$$

To model sequential composition, a special channel `done` is introduced, on which processes signal their termination. Channel `done` is used by the auxiliary operators

$$(\textsc{Assign-Op})\quad \frac{}{\langle X := E, s\rangle \rightarrow \langle \mathtt{nil}, s[s(E)/X]\rangle}$$

$$(\textsc{Seq-Op1})\quad \frac{\langle C, s\rangle \rightarrow \langle C', s'\rangle}{\langle C; D, s\rangle \rightarrow \langle C'; D, s'\rangle} \qquad (\textsc{Seq-Op2})\quad \frac{}{\langle \mathtt{nil}; D, s\rangle \rightarrow \langle D, s\rangle}$$

$$(\textsc{Cond-Op1})\quad \frac{s(E) = tt}{\langle \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ D, s\rangle \rightarrow \langle C, s\rangle}$$

$$(\textsc{Cond-Op2})\quad \frac{s(E) \neq tt}{\langle \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ D, s\rangle \rightarrow \langle D, s\rangle}$$

$$(\textsc{While-Op1})\quad \frac{s(E) = tt}{\langle \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle \rightarrow \langle C; \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle}$$

$$(\textsc{While-Op2})\quad \frac{s(E) \neq tt}{\langle \mathtt{while}\ E\ \mathtt{do}\ C, s\rangle \rightarrow \langle \mathtt{nil}, s\rangle}$$

$$(\textsc{ParL-Op1})\quad \frac{\langle C, s\rangle \rightarrow \langle C', s'\rangle}{\langle C \parallel D, s\rangle \rightarrow \langle C' \parallel D, s'\rangle} \qquad (\textsc{ParL-Op2})\quad \frac{}{\langle \mathtt{nil} \parallel D, s\rangle \rightarrow \langle D, s\rangle}$$

$$(\textsc{ParR-Op1})\quad \frac{\langle D, s\rangle \rightarrow \langle D', s'\rangle}{\langle C \parallel D, s\rangle \rightarrow \langle C \parallel D', s'\rangle} \qquad (\textsc{ParR-Op2})\quad \frac{}{\langle C \parallel \mathtt{nil}, s\rangle \rightarrow \langle C, s\rangle}$$

Fig. 2. Operational Semantics of PARIMP

Done, *Before* and *Par*, defined as follows, assuming d, d_1, d_2 to be new names:

$$Done \overset{\text{def}}{=} \overline{\mathtt{done}}.\,\mathbf{0}$$

$$C\ Before\ D \overset{\text{def}}{=} (C[d/\mathtt{done}] \mid d.\,D)\backslash d$$

$$C_1\ Par\ C_2 \overset{\text{def}}{=} (C_1[d_1/\mathtt{done}] \mid C_2[d_2/\mathtt{done}] \mid (d_1.\,d_2.Done + d_2.\,d_1.Done))\backslash\{d_1, d_2\}$$

The translation of commands is then given by:

$$\begin{aligned}
&[\![\texttt{nil}]\!] = \mathit{Done}\\
&[\![X := E]\!] = [\![E]\!]\ \mathit{Into}\,(x)\ (\overline{put_X}\langle x\rangle .\ \mathit{Done})\\
&[\![C\,;\,D]\!] = [\![C]\!]\ \mathit{Before}\ [\![D]\!]\\
&[\![(\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2)]\!] = [\![E]\!]\ \mathit{Into}\,(x)\ \ (\texttt{if}\ x\ \texttt{then}\ [\![C_1]\!]\ \texttt{else}\ [\![C_2]\!])\\
&[\![(\texttt{while}\ E\ \texttt{do}\ C)]\!] = W,\ \text{ where } W \stackrel{\text{def}}{=}\ [\![E]\!]\ \mathit{Into}\,(x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\texttt{if}\ x\ \texttt{then}\ \ [\![C]\!]\ \mathit{Before}\ W\ \texttt{else}\ \mathit{Done})\\
&[\![(C_1 \parallel C_2)]\!] = [\![C_1]\!]\ \mathit{Par}\ [\![C_2]\!]
\end{aligned}$$

Finally, the translation of a well-formed configuration $\langle C, s\rangle$ is defined by:

$$[\![\langle C, s\rangle]\!] = (\,[\![C]\!] \mid [\![s]\!]\,) \setminus \mathit{Acc}_s \cup \{\texttt{done}\}$$

where $\mathit{Acc}_s \stackrel{\text{def}}{=} \{\, get_X,\ put_X \ \mid\ X \in \texttt{dom}(s)\,\}$ is the *access sort* of state s.
As noted by Milner in [18], the above translation does not preserve the atomicity of assignment statements. Consider the program $C = (X := X + 1 \parallel X := X + 1)$. The translation of C is:

$$\begin{aligned}
[\![C]\!]\ =\ &(\,(get_X(x).\,\overline{\texttt{res}}\langle x+1\rangle\ \mid\ \texttt{res}(y).\,\overline{put_X}\langle y\rangle .\,\overline{d_1})\setminus\texttt{res}\\
&\mid\ (get_X(x).\,\overline{\texttt{res}}\langle x+1\rangle\ \mid\ \texttt{res}(y).\,\overline{put_X}\langle y\rangle .\,\overline{d_2})\setminus\texttt{res}\\
&\mid\ (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\setminus\{d_1, d_2\}
\end{aligned}$$

Here the second get_X action may be executed before the first $\overline{put_X}$ action. This means that the same value v_0 may be read for X in both assignments, and thus the same value $v_1 = v_0 + 1$ may be assigned twice to X. Hence, while C only produces the final value $v_2 = v_0 + 2$ for X, $[\![C]\!]$ may also produce the final value $v_1 = v_0 + 1$.

It is then easy to see that the translation does not preserve security. Let $C_L = (X_L := X_L + 1 \parallel X_L := X_L + 1)$ and $D_L = (X_L := X_L + 1\,;\, X_L := X_L + 1)$. Let now $\widehat{C} = (\texttt{if}\ z_H = 0\ \texttt{then}\ C_L\ \texttt{else}\ D_L)$. Then $\widehat{C}$ is secure, but $[\![\widehat{C}]\!]$ is not.

It may be shown with similar examples (see [6]) that, in order for the translation to preserve security, it should also forbid the overlapping of assignments to different variables, as well as the overlapping of assignments with expression evaluation. To prevent such overlappings, we introduce a global semaphore for the whole store:

$$\mathit{Sem} \stackrel{\text{def}}{=} \texttt{lock}.\,\texttt{unlock}.\,\mathit{Sem}$$

Correspondingly, $\overline{\texttt{lock}}$ and $\overline{\texttt{unlock}}$ actions must be introduced in the translation of the assignment, conditional and loop commands.

The translation of the assignment command becomes:

$$[\![X := E]\!]\ =\ \overline{\texttt{lock}}.\,[\![E]\!]\ \mathit{Into}\,(x)\ (\overline{put_X}\langle x\rangle .\,\overline{\texttt{unlock}}.\ \mathit{Done})$$

To introduce locks in the translation of conditionals and loops, we examine two different solutions.

Solution 1. The first revised translation of conditionals and loops is given by:

$$[\![(\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2)]\!] = \overline{\texttt{lock}}.\,[\![E]\!]\,\mathit{Into}\,(x)\,(\,\texttt{if } x\,\texttt{then}\;\overline{\texttt{unlock}}.\,[\![C_1]\!] \\ \texttt{else}\;\overline{\texttt{unlock}}.\,[\![C_2]\!]\,)$$

$$[\![(\texttt{while } E \texttt{ do } C)]\!] = W,\ \text{where } W \overset{\text{def}}{=} \overline{\texttt{lock}}.\,[\![E]\!]\,\mathit{Into}\,(x) \\ (\texttt{if } x \texttt{ then } \overline{\texttt{unlock}}.\,[\![C]\!]\ \mathit{Before}\ W \texttt{ else } \overline{\texttt{unlock}}.\,\mathit{Done})$$

Our first translation of PARIMP into CCS, based on Solution 1, is summarised in Figure 3. This is essentially a simpler version of the translation proposed by Focardi, Rossi and Sabelfeld in [9], where in addition a special `tick` action was inserted in the encoding of each statement (just before the $\overline{\texttt{unlock}}$ action, in the above encodings), so as to recover a direct correspondence between execution steps in $\langle C, s\rangle$ and their simulation in $[\![\langle C, s\rangle]\!]$. This was needed to obtain a full abstraction result.

Solution 2. The second revised translation of conditionals and loops is based on the idea of *localising the use of locks*, by using the $\overline{\texttt{lock}}$ and $\overline{\texttt{unlock}}$ actions as delimiters around the translation of expression evaluation.
Let $\mathit{getseq}_{\tilde{X}}(\tilde{x})$ be an abbreviation for $\mathit{get}_{X_1}(x_1).\cdots.\mathit{get}_{X_n}(x_n)$, and $f(\tilde{x})$ stand for $f(x_1,\ldots,x_n)$, as usual.

The *atomic translation* of expression E, denoted $[\![E]\!]_{at}$, is defined as follows:

$$[\![F(X_1,\ldots,X_n)]\!]_{at} = \overline{\texttt{lock}}.\,\mathit{getseq}_{\tilde{X}}(\tilde{x}).\,\overline{\texttt{res}}\langle f(\tilde{x})\rangle.\,\overline{\texttt{unlock}}.\,\mathbf{0}$$

The translation of conditionals and loops is then adapted by replacing $[\![E]\!]$ by $[\![E]\!]_{at}$:

$$[\![(\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2)]\!] = [\![E]\!]_{at}\ \mathit{Into}\,(x)\ (\texttt{if } x \texttt{ then } [\![C_1]\!] \texttt{ else } [\![C_2]\!])$$

$$[\![(\texttt{while } E \texttt{ do } C)]\!] = W,\ \text{where } W \overset{\text{def}}{=} [\![E]\!]_{at}\ \mathit{Into}\,(x) \\ (\texttt{if } x \texttt{ then } [\![C]\!]\ \mathit{Before}\ W \texttt{ else } \mathit{Done})$$

The second translation of PARIMP into CCS, based on Solution 2, is given in Figure 4, where for clarity we denote the translation by the symbol $\langle\!\!\langle\ \rangle\!\!\rangle$ instead of $[\![\ ]\!]$. In the rest of the paper, we shall concentrate on the first translation, but all our results also hold for the second translation.

3.3 The translation preserves security

In this section we show that the translation just described preserves security. This result will be based, as usual, on an operational correspondence between programs (or more exactly, configurations) in the source language and their images in the target language. In order to relate the behaviour of a configuration $\langle C, s\rangle$ with that of its image $[\![\langle C, s\rangle]\!] = (\,[\![C]\!] \mid [\![s]\!] \mid \mathit{Sem}\,) \setminus \mathit{Acc}_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\}$, we must provide a means to observe the changes performed by $[\![C]\!]$ on $[\![s]\!]$ in CCS[3]. To

[3] Note that, as it stands, the translation maps any configuration $\langle C, s\rangle$ to an unobservable CCS process.

Semaphore:

$$Sem \overset{\text{def}}{=} \mathtt{lock}.\,\mathtt{unlock}.\,Sem$$

Translation of states:

$$[\![s]\!] \;=\; Reg_{X_1}(s(X_1)) \mid \cdots \mid Reg_{X_n}(s(X_n)) \qquad \text{if } \mathtt{dom}(s) = \{X_1, \ldots, X_n\}$$

Translation of expressions:

$$[\![F(X_1, \ldots, X_n)]\!] \;=\; get_{X_1}(x_1).\cdots.get_{X_n}(x_n).\,\overline{\mathtt{res}}\langle f(x_1, \ldots, x_n)\rangle.\,\mathbf{0}$$

Translation of commands:

$$\begin{array}{rcl}
[\![\mathtt{nil}]\!] &=& Done \\
[\![X := E]\!] &=& \overline{\mathtt{lock}}.\,[\![E]\!]\; Into\,(x)\; (\overline{put_X}\langle x\rangle.\,\overline{\mathtt{unlock}}.\,Done) \\
[\![C\,;\,D]\!] &=& [\![C]\!]\; Before\,[\![D]\!] \\
[\![(\mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2)]\!] &=& \overline{\mathtt{lock}}.\,[\![E]\!]\; Into\,(x)\;\; (\,\mathtt{if}\ \ x\ \mathtt{then}\ \overline{\mathtt{unlock}}.\,[\![C_1]\!] \\
&& \qquad\qquad\qquad\qquad\quad \mathtt{else}\ \overline{\mathtt{unlock}}.\,[\![C_2]\!]\,) \\
[\![(\mathtt{while}\ E\ \mathtt{do}\ C)]\!] &=& W,\ \text{where } W \overset{\text{def}}{=} \overline{\mathtt{lock}}.\,[\![E]\!]\; Into\,(x) \\
&& (\mathtt{if}\ x\ \mathtt{then}\ \overline{\mathtt{unlock}}.\,[\![C]\!]\; Before\ W\ \mathtt{else}\ \overline{\mathtt{unlock}}.\,Done)
\end{array}$$

Translation of well-formed configurations $\langle C, s\rangle$:

$$[\![\langle C, s\rangle]\!] = (\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \setminus Acc_s \,\cup\, \{\mathtt{done}, \mathtt{lock}, \mathtt{unlock}\}$$

Access sort of a state s:

$$Acc_s \overset{\text{def}}{=} \{\, get_X,\, put_X \;\mid\; X \in \mathtt{dom}(s)\,\}$$

Fig. 3. Translation of PARIMP into CCS

this end we introduce, as in [25] and [9], special channels dedicated to the exchange of data between processes and the environment, which we call *in* and *out*: the environment uses channel in_X to feed a new value into register Reg_X, and channel out_X to retrieve the current value of Reg_X.

The definition of registers is then adapted to account for the new actions. Each Reg_X in our translation is replaced by the *observable register* $OReg_X$ defined by:

$$\begin{array}{rl}
OReg_X(v) \overset{\text{def}}{=} & put_X(x).\,OReg_X(x) \;+\; \overline{get_X}\langle v\rangle.\,OReg_X(v) \;+ \\
& \overline{\mathtt{lock}}.\,(\,in_X(x).\,\overline{\mathtt{unlock}}.\,OReg_X(x) \;+\; \overline{\mathtt{unlock}}.\,OReg_X(v)\,) \;+ \\
& \overline{\mathtt{lock}}.\,(\,\overline{out_X}\langle v\rangle.\,\overline{\mathtt{unlock}}.\,OReg_X(v) \;+\; \overline{\mathtt{unlock}}.\,OReg_X(v)\,)
\end{array}$$

Here the locks around the $in_X(x)$ and $\overline{out_X}\langle v\rangle$ prefixes are used to prevent the environment from accessing the register while this is being used by some process. Note that after committing to communicate with the environment by means of a $\overline{\mathtt{lock}}$ action, an observable register can always withdraw its commitment by doing an $\overline{\mathtt{unlock}}$ action, and get back to its initial state. This ensures that the pool of observable registers, when run in parallel with the semaphore, is always weakly bisimilar to the image of some state s.

Notation: Let Env be the set $\{in_X, out_X \mid X \in Var\}$. We define then the set of *environmental actions* to be $Act_{Env} \overset{\text{def}}{=} \{\alpha \in Act \mid subj(\alpha) \in Env\}$.

Semaphore:

$$Sem \overset{\text{def}}{=} \mathtt{lock}.\,\mathtt{unlock}.\,Sem$$

Translation of states:

$$\langle\!\lbrack s \rbrack\!\rangle = Reg_{X_1}(s(X_1)) \mid \cdots \mid Reg_{X_n}(s(X_n)) \qquad \text{if } \mathtt{dom}(s) = \{X_1, \ldots, X_n\}$$

Translation of expressions:

$$\langle\!\lbrack F(X_1, \ldots, X_n) \rbrack\!\rangle = \underbrace{get_{X_1}(x_1). \cdots . get_{X_n}(x_n)}_{getseq_{\vec{X}}(\vec{x})}.\,\overline{\mathtt{res}}\langle \underbrace{f(x_1, \ldots, x_n)}_{f(\vec{x})} \rangle.\,\mathbf{0}$$

Atomic translation of expressions:

$$\langle\!\lbrack F(X_1, \ldots, X_n) \rbrack\!\rangle_{at} = \overline{\mathtt{lock}}.\,getseq_{\vec{X}}(\vec{x}).\,\overline{\mathtt{res}}\langle f(\vec{x}) \rangle.\,\overline{\mathtt{unlock}}.\,\mathbf{0}$$

Translation of commands:

$$\begin{aligned}
\langle\!\lbrack \mathtt{nil} \rbrack\!\rangle &= Done \\
\langle\!\lbrack X := E \rbrack\!\rangle &= \overline{\mathtt{lock}}.\,\langle\!\lbrack E \rbrack\!\rangle\ Into\,(x)\ (\overline{put_X}\langle x \rangle.\,\overline{\mathtt{unlock}}.\,Done) \\
\langle\!\lbrack C\,;\,D \rbrack\!\rangle &= \langle\!\lbrack C \rbrack\!\rangle\ Before\ \langle\!\lbrack D \rbrack\!\rangle \\
\langle\!\lbrack (\mathtt{if}\ E\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2) \rbrack\!\rangle &= \langle\!\lbrack E \rbrack\!\rangle_{at}\ Into\,(x)\ \ (\mathtt{if}\ x\ \mathtt{then}\ \langle\!\lbrack C_1 \rbrack\!\rangle\ \mathtt{else}\ \langle\!\lbrack C_2 \rbrack\!\rangle) \\
\langle\!\lbrack (\mathtt{while}\ E\ \mathtt{do}\ C) \rbrack\!\rangle &= W,\ \text{where } W \overset{\text{def}}{=} \langle\!\lbrack E \rbrack\!\rangle_{at}\ Into\,(x) \\
&\qquad (\mathtt{if}\ x\ \mathtt{then}\ \langle\!\lbrack C \rbrack\!\rangle\ Before\ W\ \mathtt{else}\ Done) \\
\langle\!\lbrack (C_1 \parallel C_2) \rbrack\!\rangle &= \langle\!\lbrack C_1 \rbrack\!\rangle\ Par\ \langle\!\lbrack C_2 \rbrack\!\rangle
\end{aligned}$$

Translation of well-formed configurations $\langle C, s \rangle$:

$$\langle\!\lbrack \langle C, s \rangle \rbrack\!\rangle = (\ \langle\!\lbrack C \rbrack\!\rangle \mid \langle\!\lbrack s \rbrack\!\rangle \mid Sem\,) \setminus Acc_s \cup \{\mathtt{done}, \mathtt{lock}, \mathtt{unlock}\}$$

Access sort of a state s:

$$Acc_s \overset{\text{def}}{=} \{\, get_X,\, put_X \ \mid\ X \in \mathtt{dom}(s)\,\}$$

Fig. 4. Alternative translation of PARIMP into CCS

As in [9], we define now labelled transitions $\xrightarrow{in_X v}$ and $\xrightarrow{\overline{out_X v}}$ for configurations[4]:

$$(\textsc{In-Op}) \quad \frac{X \in \mathtt{dom}(s)}{\langle C, s\rangle \xrightarrow{in_X v} \langle C, s[v/X]\rangle} \qquad (\textsc{Out-Op}) \quad \frac{s(X) = v}{\langle C, s\rangle \xrightarrow{\overline{out_X v}} \langle C, s\rangle}$$

We also extend τ-transitions to configurations by letting:

$$\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle \quad \Leftrightarrow_{\text{def}} \quad \langle C, s\rangle \rightarrow \langle C', s'\rangle$$

We may now define weak labelled transitions $\langle C, s\rangle \overset{\alpha}{\Longrightarrow} \langle C', s'\rangle$ on configurations, where $\alpha \in \mathit{Act}_{\mathit{Env}} \cup \{\tau\}$, exactly in the same way as for CCS processes.

The operational correspondence between well-formed configurations $\langle C, s\rangle$ and their images in CCS is then given by the following two Lemmas:

Lemma 3.5 (Program transitions are preserved by the translation) *Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in \mathit{Act}_{\mathit{Env}}$. Then:*

(i) *If $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$, then there exists P such that $[\![\langle C, s\rangle]\!] \overset{\alpha}{\Longrightarrow} P = [\![\langle C', s'\rangle]\!]$*

(ii) *If $\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle$, then there exists P such that $[\![\langle C, s\rangle]\!] \overset{\tau}{\Longrightarrow} P \approx [\![\langle C', s'\rangle]\!]$*

Lemma 3.6 (Process transitions are reflected by the translation) *Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in \mathit{Act}_{\mathit{Env}}$. Then:*

(i) *If $[\![\langle C, s\rangle]\!] \overset{\alpha}{\Longrightarrow} P$, then there exist C', s' such that $\langle C, s\rangle \overset{\alpha}{\Longrightarrow} \langle C', s'\rangle$ and $P \approx [\![\langle C', s'\rangle]\!]$.*

(ii) *If $[\![\langle C, s\rangle]\!] \overset{\tau}{\Longrightarrow} P$, then either $[\![\langle C, s\rangle]\!] \approx P$ or there exist C', s' such that $\langle C, s\rangle \overset{\tau}{\Longrightarrow} \langle C', s'\rangle$ and $P \approx [\![\langle C', s'\rangle]\!]$.*

The proof of Lemma 3.5 is rather straightforward, while that of Lemma 3.6 is much more elaborate. Intuitively, we need to decompose a computation $[\![\langle C, s\rangle]\!] \overset{\alpha}{\Longrightarrow} P$ or $[\![\langle C, s\rangle]\!] \overset{\tau}{\Longrightarrow} P$ into a sequence of *micro-computations*, each of which is the simulation of a single (or empty) step of the source configuration $\langle C, s\rangle$, possibly interspersed with *relay moves* (parallel moves which do not affect the state and thus can be interleaved with *transactions*, which are sequences of moves accessing the state). The proof, as well as the auxiliary definitions and results required for it, may be found in the full version of the paper [6].

To compare the notion of L-security on PARIMP with that of PBNDC on CCS, it is convenient to characterise L-bisimilarity on programs by means of a bisimilarity up to high on configurations, following [9]. To this end, we introduce *restricted configurations* of the form $\langle C, s\rangle\backslash R$, where $\langle C, s\rangle$ is well-formed and $R \subseteq \mathit{Env}$, whose semantics is specified by the rule:

$$(\textsc{Res-Op}) \quad \frac{\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle \qquad \mathit{subj}(\alpha) \notin R}{\langle C, s\rangle\backslash R \xrightarrow{\alpha} \langle C', s'\rangle\backslash R}$$

Let *ResConf* denote the set of restricted configurations, ranged over by *cfg*, *cfg*$'$, *cfg*$_i$. For any $\mathit{cfg} = \langle C, s\rangle\backslash R \in \mathit{ResConf}$, we let $\mathtt{dom}(\mathit{cfg}) \overset{\text{def}}{=} \mathtt{dom}(s)$.

[4] From now on we use v, v' also for PARIMP values, assuming them to coincide with CCS values.

We extend our translation to restricted configurations by letting:

$$[\![\langle C, s\rangle \backslash R]\!] \stackrel{\text{def}}{=} [\![\langle C, s\rangle]\!] \backslash R$$

Let now $Env_H \stackrel{\text{def}}{=} \{in_X, out_X \mid X \in H\}$ and $Env_L \stackrel{\text{def}}{=} \{in_X, out_X \mid X \in L\}$. We introduce a transition relation $\overset{\alpha}{\longmapsto}_H$ on *ResConf*, for any $\alpha \in Act_{Env} \cup \{\tau\}$:

$$cfg \overset{\alpha}{\longmapsto}_H cfg' \stackrel{\text{def}}{=} \begin{cases} cfg \xrightarrow{\alpha} cfg' \text{ or } cfg = cfg' & \text{if } subj(\alpha) \in Env_H \cup \{\tau\} \\ cfg \xrightarrow{\alpha} cfg' & \text{if } subj(\alpha) \in Env_L \end{cases}$$

The transition relation $\overset{\tilde{\alpha}}{\longmapsto}_H$ is used to define the following notion of bisimulation on restricted configurations:

Definition 3.7 [Bisimulation up-to-high on configurations]

A symmetric relation $\mathcal{S} \subseteq (ResConf \times ResConf)$ is a *bisimulation up to high* if $cfg_1 \; \mathcal{S} \; cfg_2$ implies $\mathtt{dom}(cfg_1) = \mathtt{dom}(cfg_2)$ and, for any $\alpha \in Act_{Env} \cup \{\tau\}$:

If $cfg_1 \xrightarrow{\alpha} cfg_1'$ then there exists cfg_2' such that $cfg_2 \overset{\alpha}{\longmapsto}_H cfg_2'$ and $cfg_1' \; \mathcal{S} \; cfg_2'$.

Two configurations cfg_1, cfg_2 are *bisimilar up to high*, written $cfg_1 \sim_H cfg_2$, if $cfg_1 \; \mathcal{S} \; cfg_2$ for some bisimulation up to high $\mathcal{S}$.

This leads us to the following characterisations of L-bisimilarity and L-security:

Proposition 3.8 (Characterisation of $\simeq_L$ in terms of $\sim_H$)

Let C, D be PARIMP *programs. Then $C \simeq_L D$ if and only if for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed, we have $\langle C, s\rangle \sim_H \langle D, s\rangle \backslash Env_H$ and $\langle C, s\rangle \backslash Env_H \sim_H \langle D, s\rangle$.*

Corollary 3.9 (Alternative characterisation of L-security)

Let C be a PARIMP program. Then C is L-secure if and only if $\langle C, s\rangle \sim_H \langle C, s\rangle \backslash Env_H$ for any s such that $\langle C, s\rangle$ is well-formed.

Suppose that channels get_X and put_X have the same security level as variable X, that channels $\mathtt{lock}, \mathtt{unlock}, \mathtt{res}$ and $\mathtt{done}$ have level h, and that renaming preserves security levels. We may then show that the translation preserves security.

Theorem 3.10 (Security is preserved by the translation)

If C is a L-secure program, then for any state s such that $\langle C, s\rangle$ is well-formed, $[\![\langle C, s\rangle]\!]$ satisfies $\mathsf{PBNDC}_{\mathcal{H}}$, where $\mathcal{H} \stackrel{\text{def}}{=} \{get_X, put_X, in_X, out_X \mid X \in H\} \cup \{\mathtt{lock}, \mathtt{unlock}, \mathtt{res}, \mathtt{done}\}$.

As a by-product, we show that the translation preserves the behavioural equivalence which is obtained from low bisimilarity by assuming all program variables to be low, and therefore observable.

If $H = \emptyset$, it is easy to see that L-bisimilarity reduces to the following:

Definition 3.11 [Behavioural equivalence on programs]

A symmetric relation $\mathcal{S} \subseteq (\mathcal{C} \times \mathcal{C})$ is a *program bisimulation* if $C\,\mathcal{S}\,D$ implies, for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed:

$$\text{If } \langle C, s\rangle \rightarrow \langle C', s'\rangle, \text{ then there exists } D' \text{ such that } \langle D, s\rangle \mapsto \langle D', s'\rangle \text{ and } C'\,\mathcal{S}\,D'.$$

Two programs C and D are *behaviourally equivalent*, written $C \simeq D$, if $C\,\mathcal{S}\,D$ for some program bisimulation $\mathcal{S}$.

Similarly, the transitions $\overset{\alpha}{\longmapsto}_H$ and the bisimilarity $\sim_H$ on configurations reduce to the transitions $\overset{\alpha}{\longmapsto}$ and the bisimilarity $\sim$ defined as follows:

$$cfg \overset{\alpha}{\longmapsto} cfg' \quad \overset{\text{def}}{=} \quad \begin{cases} cfg \xrightarrow{\alpha} cfg' \text{ or } cfg = cfg' & \text{if } \alpha = \tau \\ cfg \xrightarrow{\alpha} cfg' & \text{otherwise} \end{cases}$$

Definition 3.12 [Bisimulation on configurations]
A symmetric relation $\mathcal{S}$ on configurations is a *bisimulation* if $\langle C, s\rangle\ \mathcal{S}\ \langle D, t\rangle$ implies $s = t$ and, for any $\alpha \in Act_{Env} \cup \{\tau\}$:

$$\text{If } \langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle, \text{ then there exists } D' \text{ such that } \langle D, t\rangle \overset{\alpha}{\longmapsto} \langle D', t'\rangle \text{ and } \langle C', s'\rangle\ \mathcal{S}\ \langle D', t'\rangle.$$

Then $\langle C, s\rangle$ and $\langle D, t\rangle$ are *bisimilar*, written $\langle C, s\rangle \sim \langle D, t\rangle$, if $\langle C, s\rangle\ \mathcal{S}\ \langle D, t\rangle$ for some bisimulation $\mathcal{S}$.

Proposition 3.13 (Characterisation of $\simeq$ in terms of $\sim$)
Let C, D be programs. Then $C \simeq D$ if and only if for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed, we have $\langle C, s\rangle \sim \langle D, s\rangle$.

Theorem 3.14 (Behavioural equivalence is preserved by the translation)

If $C \simeq D$, then for any state s such that $\langle C, s\rangle$ is well-formed, $[\![\langle C, s\rangle]\!] \approx [\![\langle D, s\rangle]\!]$.

Note that the equivalence $\simeq$ is rather intensional. For instance it does not in general equate the two programs `nil`$; P$ and P. On the other hand we have P ; `nil` $\simeq P$, as well as (`nil` $\parallel P) \simeq P$. Indeed, as soon as two programs stop modifyng the memory, they are identified by $\simeq$. This is not surprising since $\simeq$ is derived from $\simeq_L$, which was designed to capture precisely this property as regards the low memory. However, we may wish to slightly relax $\simeq$ so as to obtain a more natural behavioural equivalence $\cong$ on PARIMP, such that `nil`$; P \cong P$. This can be obtained as follows.

Let $\langle C, s\rangle \rightsquigarrow \langle C', s'\rangle$ denote the *administrative transition* relation obtained by using only the subset of rules (Seq-Op2), (ParL-Op1), (ParL-Op2), (ParR-Op1) and (ParR-Op2) of Figure 2. We may then define an equivalence $\cong$ which always identifies programs whose behaviours differ only for administrative moves.

Definition 3.15 [`nil`-insensitive behavioural equivalence on programs]
A symmetric relation $\mathcal{S} \subseteq (\mathcal{C} \times \mathcal{C})$ is a `nil`-*insensitive program bisimulation* if $C\,\mathcal{S}\,D$ implies, for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed:

$$\text{If } \langle C, s\rangle \rightarrow \langle C', s'\rangle, \text{ then there exists } D' \text{ such that } \langle D, s\rangle \rightsquigarrow^{*}\mapsto \rightsquigarrow^{*} \langle D', s'\rangle$$
$$\text{and } \quad C' \,\mathcal{S}\, D'.$$

The nil-*insensitive behavioural equivalence* $\cong$ is then defined by: $C \cong D$ if $C \,\mathcal{S}\, D$ for some nil-insensitive program bisimulation $\mathcal{S}$.

The definition of L-bisimulation and L-security could be weakened in a similar way. We conjecture that all the results proved in this section would easily extend to the nil-insensitive notions of L-security and behavioural equivalence.

Finally, we may wonder whether the arrow $\mapsto$ could be replaced by the arrow $\longrightarrow^{*}$ in the definition of L-bisimulation (and behavioural equivalence), while preserving our results. An advantage of this choice would be to open the possibility for a full abstraction result (restricted to the processes obtained as images of programs), since the resulting security property, which is that proposed by Sabelfeld and Sands in [27], is fully time-insensitive and thus very close in spirit to weak bisimulation.

Indeed, it is easy to see that our translation is not fully abstract (with respect to any of our security properties, whether nil-sensitive or nil-insensitive). For instance the process of Example 3.4 is not secure nor nil-insensitive secure, while its encoding into CCS is secure. For the nil-sensitive security property, another reason of failure for full abstraction is illustrated by the program:

$$C = (\texttt{if } X_H = 0 \texttt{ then } Y_L := 0 \texttt{ else nil}\,;\, Y_L := 0)$$

Here $[\![C]\!]$ is secure while C is not secure (however it is nil-insensitive secure).

At this point of discussion, it may seem surprising that a full abstraction result could be obtained in [9] for a time-sensitive security notion, which is stronger than our security properties and thus further away from weak bisimulation. In fact, as mentioned earlier, this full abstraction result was obtained by using special tick actions in the translation, whose function was precisely to enforce a correspondence between steps in the source program and their encodings in the target process. Indeed, it is easy to see that full abstraction would fail for time-sensitive security in the absence of such tick actions. For consider the program:

$$C' = (\texttt{if } X_H = 0 \texttt{ then loop else nil})$$

Both C' and the program C above are insecure with respect to the time-sensitive security property of [9]. On the other hand, the encodings of these programs without tick actions (i.e. according to the translation of Figure 3), are secure.

The question of whether our results could be extended to the security property of Sabelfeld and Sands [27], and whether a “natural” full abstraction result could be obtained in that case, is left for further investigation.

3.4 The translation does not preserve security types

In this section, we show that the type system presented in Section 2 is not reflected by the translation of Figure 3 (nor by that of Figure 4). We then sketch a solution to overcome this problem.

Consider the program $C = (X_H := X_H + 1\,;\, Y_L := Y_L + 1)$, which is typable in the type systems of e.g. [30,27,29,4]. This program is translated to the process $[\![C]\!]$:

$$(\nu d)\,(\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_1)\,(get_{X_H}(x).\,\overline{\texttt{res}_1}\langle x+1\rangle \mid \texttt{res}_1(z_1).\,\overline{put_{X_H}}\langle z_1\rangle.\,\overline{\texttt{unlock}}.\,\overline{d}) \;\mid$$
$$d.\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_2)\,(get_{Y_L}(y).\,\overline{\texttt{res}_2}\langle y+1\rangle \mid \texttt{res}_2(z_2).\,\overline{put_{Y_L}}\langle z_2\rangle.\,\overline{\texttt{unlock}}.\,\overline{\texttt{done}})\,)$$

Now, it is easy to see that $[\![C]\!]$ is not typable in the security type system of Section 2. Indeed, there is no assignment of security levels for the channels `lock`, `unlock` and d which allows $[\![C]\!]$ to be typed. First of all, note that channels `lock` and `unlock` must have the same security level since each of them follows the other in the semaphore (and in the registers). Consider now the two top parallel components of $[\![C]\!]$: for the first component to be typable, `unlock` and d should be high (and thus `lock` should be high too); for the second component to be typable, d and `lock` should be low (and thus `unlock` should be low too). In other words, the two components impose conflicting constraints on the levels of channels `lock`, `unlock` and d.

A possible solution to this problem is to relax the type system by treating more liberally channels like `lock`, `unlock` and d (and hence `done`, since d is obtained from `done` by renaming), which carry no values and are restricted. The idea, borrowed from previous work [16,14,15,33,17], is that actions on these channels are *data flow irrelevant* insofar as they are guaranteed to occur, since in this case their occurrence does not bring any information. The typing rule (Sum) may then be made less restrictive for these actions, while keeping their security level to h. It can be observed that replacing rule (Sum) by the rule (Sum-Lax) discussed at page 8 would not solve the problem.

Note that action $\overline{\texttt{lock}}$ is eventually enabled from any state of $[\![\langle C, s\rangle]\!]$, since the semaphore and the pool of registers cyclically come back to their initial state. The situation is not as simple as concerns the channel `done`, or more precisely one of its renamings d, as the occurrence of the (unique) complementary action $\overline{d}$ could be prevented by divergence or deadlock. Notice however that deadlock cannot arise in a process $[\![\langle C, s\rangle]\!]$, because the source configuration $\langle C, s\rangle$ can only contain livelocks, due to busy waiting and thus to while loops. Now, by imposing restrictions on the use of loops and conditionals in programs (as proposed in [30,27,29,4]), one may either enforce the occurrence of $\overline{d}$ in their images, or make sure that if this occurrence is uncertain because of some high test, then no low memory change can depend on it. In conclusion, provided the set of source programs is appropriately restricted by typing, the lock channels and the *relay channels* obtained by renaming channel `done` can be safely be given level h.

The formalisation of a type system along these lines, as well as the study of a more general security type system for state-oriented noninterference on CCS, is the subject of current work.

4 Conclusion and related work

We addressed the question of relating language-based and process-based security, by focussing on a simple parallel imperative language à la Volpano and Smith [30] and on Milner's calculus CCS [18]. We presented an encoding from the former to the latter, essentially a variant of Milner's well-known translation, and showed that it preserves a time-insensitive security property. In doing so, we extended previous work by Focardi, Rossi and Sabelfeld [9] in several respects: (1) we considered a parallel rather than a sequential language, (2) we studied a time-insensitive rather than a time-sensitive security property, (3) we examined two variants of Milner's translation, which are both simpler than that used in [9], and (4) we proposed a security type system for PBNDC on CCS which, although failing to reflect a security type system for the source language, appears to be a good step towards that purpose.

As concerns related work, besides the paper [25] by Mantel and Sabelfeld, who were the first to establish security-preserving translations between programming languages and specification formalisms, we should mention the thorough comparison of language-based and process-based security carried out in [14] by Honda and Yoshida, who proposed type-preserving embeddings of powerful languages, both imperative and functional, into a variant of the asynchronous π-calculus. Closely related to [14] is Kobayashi's security type system [17], which is equipped with a type inference algorithm. In both cases, the process calculus is more expressive than CCS and the type system is rather complex, as it is meant to grant both a security property and other correctness properties. As regards the expressiveness of the considered languages, our work is clearly less ambitious than [14]. However, an advantage of focussing on a first-order process calculus which does not require any classical typing, is that the typing requirements for security may be clearly isolated. Moreover, some issues related to atomicity and to the impact of the sum operator on security, arise in CCS but not in the asynchronous π-calculus.

Acknowledgments

I would like to thank Maria-Grazia Vigliotti for her contribution at an early stage of this work, Frédéric Boussinot for helpful remarks, and the anonymous referees for useful feedback on the submitted version.

References

[1] Johan Agat. Transforming out timing leaks. *Proceedings of POPL '00*, ACM Press, pages 40–53, 2000.

[2] A. Almeida Matos, G. Boudol and I. Castellani. Typing noninterference for reactive programs. *Journal of Logic and Algebraic Programming* 72: 124-156, 2007.

[3] G. Barthe and L. Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proceedings of FMSE'04*, 2004.

[4] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science* 281(1): 109-130, 2002.

[5] A. Bossi, R. Focardi, C. Piazza and S. Rossi. Verifying persistent security properties. *Computer Languages, Systems and Structures* 30(3-4): 231-258, 2004.

[6] I. Castellani. State-oriented noninterference for CCS (full version). INRIA Research Report, October 2007. URL: http://www-sop.inria.fr/mimosa/personnel/Ilaria.Castellani/main-publications.html.

[7] R. Focardi and S. Rossi. Information flow security in dynamic contexts. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[8] R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In *Foundations of Security Analysis and Design - Tutorial Lectures* (R. Focardi and R. Gorrieri, Eds.), volume 2171 of *LNCS*, Springer, 2001.

[9] R. Focardi, S. Rossi and A. Sabelfeld. Bridging Language-Based and Process Calculi Security. In *Proceedings of FoSSaCs'05*, volume 3441 of *LNCS*, Springer-Verlag, 2005.

[10] S. Crafa and S. Rossi. A theory of noninterference for the π-calculus. In *Proceedings of Symp. on Trustworthy Global Computing TGC'05*, volume 3705 of *LNCS*, Springer-Verlag, 2005.

[11] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous π-calculus. *ACM TOPLAS* 24(5): 566-591, 2002.

[12] M. Hennessy. The security π-calculus and noninterference. *Journal of Logic and Algebraic Programming* 63(1): 3-34, 2004.

[13] K. Honda, V. Vasconcelos and N. Yoshida. Secure information flow as typed process behavior. In *Proceedings of ESOP'00*, volume 1782 of *LNCS*, pages 180-199, Springer-Verlag, 2000.

[14] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of POPL'02*, ACM Press, pages 81-92. January, 2002.

[15] N. Yoshida, K. Honda and M. Berger. Linearity and bisimulation. In *Proceedings of FoSSaCs'02*, volume 2303 of *LNCS*, pages 417-433, Springer-Verlag, 2002.

[16] N. Kobayashi, B. Pierce and D. Turner. Linearity and the π-calculus. In *Proceedings of POPL'96*, ACM Press, pages 358-371, 1996.

[17] N. Kobayashi. Type-based Information Flow Analysis for the π-Calculus. *Acta Informatica* 42(4-5): 291-347, 2005.

[18] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[19] A. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of POPL'99*, ACM Press, pages 228-241, 1999.

[20] A. Myers, L. Zheng, S. Zdancewic, S. Chong and N. Nystrom. Jif: Java information flow. Software release, http://www.cs.cornell.edu/jif, 2001.

[21] F. Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

[22] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS* 25(1): 117-158, 2003.

[23] V. Simonet. The FlowCaml system: documentation and user manual. INRIA RR n. 0282, 2003.

[24] P. Ryan and S. Schneider. Process algebra and noninterference. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[25] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security* 11(4): 615–676, 2003.

[26] A. Sabelfeld and A. C. Myers, Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 211:5-19, 2003.

[27] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*, pages 200-214, 2000.

[28] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[29] G. Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 115–125, 2001.

[30] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. *Proceedings of POPL '98*, ACM Press, pages 355–364, 1998.

[31] D. Volpano and G. Smith. Probabilistic Noninterference in a Concurrent Language. *Journal of Computer Security* 7(2-3): 231–253, 1999.

[32] D. Volpano, G. Smith and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4(3):167–187, 1996.

[33] S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation* 15(2-3):209-234, 2002.