State-oriented noninterference for CCS

Ilaria Castellani

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

State-oriented noninterference for CCS

Ilaria Castellani

N° 6322

October 2007

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--6322--FR+ENG

State-oriented noninterference for CCS

Ilaria Castellani

Thème COM — Systèmes communicants
Projet Mimosa

Rapport de recherche n° 6322 — October 2007 — 58 pages

Abstract: We address the question of typing *noninterference* (NI) in Milner's Calculus of Communicating Systems (CCS), in such a way that Milner's translation of a standard parallel imperative language into CCS 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 security 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 different 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 natural behavioural equivalence on programs. We present a type system ensuring the PBNDC-property on CCS, inspired from 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.

Key-words: Noninterference, type systems, parallel imperative languages, process calculi, bisimulation.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Non-interférence orientée-états pour CCS

Résumé : Nous nous intéressons à la question du typage de la propriété de *non-interférence* (NI) dans le calcul CCS (Calculus of Communicating Systems) de Milner. Le but recherché est de prouver que la traduction de Milner d'un langage impératif parallèle vers CCS préserve à la fois une propriété de non-interférence connue et l'un des systèmes de types associés. Récemment, Focardi, Rossi et Sabelfeld ont montré qu'une variante de la traduction de Milner, restreinte au fragment séquentiel du langage, préserve une propriété de NI sensible au temps en lui faisant correspondre une propriété de sécurité existante pour CCS, appelée *Non Deductibilité Persistante par Compositions basée sur la Bisimulation* (PBNDC). Toutefois, CCS n'ayant pas été préalablement équipé d'un système de types pour la sécurité, la question de la préservation des types par la traduction ne pouvait être posée. Nous étendons le résultat de Focardi, Rossi et Sabelfeld en montrant qu'une nouvelle variante de la traduction de Milner préserve une propriété de NI *insensible au temps* sur l'ensemble du langage, en l'envoyant également sur la propriété de PBNDC. Au passage, nous formalisons un résultat appartenant au folklore, notamment que la traduction de Milner préserve une équivalence comportementale sur les programmes. Nous présentons un système de types pour CCS garantissant la propriété de PBNDC. Ce système est inspiré de systèmes de types précédemment proposés pour le π-calcul. Malheureusement, notre système de types s'avère trop restrictif pour refléter l'un des systèmes de types existants pour le langage impératif. Nous esquissons une solution à ce problème.

Mots-clés : Non-interférence, systèmes de types, langages impératifs avec parallélisme, calculs de processus, 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, 7, 13, 21, 10, 14, 11, 5, 17, 9]. In the following 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.

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 that have been proposed for process calculi, mostly based on trace equivalence, testing or bisimulation (cf [7] for a review). In general, these properties do not make a distinction between the flow of data and the flow of control: indeed, these two kinds of flow are closely intertwined in process calculi. Let us consider some examples to illustrate this point.

In the calculus CCS, an input process $a(x).P$ receives a value v on channel a and then behaves like $P\{v/x\}$. Symmetrically, an output process $\overline{a}\langle e\rangle.P$ emits the value of expression e on channel a and then behaves like 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:

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

where $done_\ell$ is a low channel without parameters and v is a constant value. Although these processes do not directly transfer data from high to low level, they are considered insecure because they can be used to implement indirect insecure flows, as in the following process (where x is assumed to be boolean and channels c_ℓ and d_ℓ are restricted and thus can be used only for internal synchronisations):

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

This process is insecure because, depending on the value received for x on the high channel get_h, it will perform two different synchronisations, followed by emissions of different values on the low channel put_ℓ. Note that the first component of P is more elaborate than the process $get_h(x).\overline{done}_\ell$ above, since after receiving a value for x it performs a test on it.

The following variants of process P, where a high input without parameters is followed by a low output, are similarly insecure:

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

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

These 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 and it is easy to find processes which do not meet it and yet satisfy noninterference. However, this criterion may serve, and indeed has been used, as a basis for defining *security type systems* for process calculi. This brings us to the question of methods and tools for ensuring NI properties.

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 that have been studied so far have been equipped with a type system or some other tool to enforce the compliance of programs with the desired security property (see for instance [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 [10, 11] 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 [9] and by Kobayashi [17]. This last work provides a sophisticated security analysis, together with a type inference algorithm for it. 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 [8], 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 [6]. However, since CCS was not equipped with a security type system, the question of type preservation could not be addressed.

Taking [8] 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 (this was a kind of folklore result). We

also propose a type system for ensuring PBNDC, inspired by the type systems of [21, 10, 11] 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 the properties of BNDC and PBNDC for CCS and we present a type system ensuring the latter. In Section 3 we review Milner's translation of a parallel imperative language into CCS, and adapt it in order to make it preserve a time-insensitive NI property, as well as a behavioural equivalence. We then show that the translation does not preserve types, and conclude with a discussion about type preservation and related work.

2 A simple security type system for CCS

In this section we present a simple security type system for the calculus CCS, similar to those proposed for the π-calculus by Pottier [21] and by Hennessy and Riely [10, 11]. We prove that this type system ensures the security property of *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), introduced by Focardi and Rossi in [6] and further studied in [5, 8, 9].

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 $val : Exp \to Val$ 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:

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

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:

$$T \quad ::= \quad A \;\mid\; (\mathtt{rec}\, A(\tilde{x})\,.\,P)$$

where P is a CCS process, whose syntax is defined below.

A term $(\mathtt{rec}\, A(\tilde{x})\,.\,P)$ 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 now given by:

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

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 $(\mathtt{rec}\; A(\tilde{x})\,.\,P)(\vec{e})$, 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 $P \setminus K$, 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:

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

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 semantics of processes is defined in Figure 1. A nondeterministic sum $\sum_{i\in I} \pi_i.P_i$ executes one of its summands $\pi_i.P_i$, 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. Finally, a recursive process behaves like its body where each occurrence of the process identifier is replaced by the process definition.

2.2 Security properties for CCS

We now review two security properties for CCS: *Bisimulation-based Non Deducibility on Compositions* (BNDC), introduced by Focardi and Gorrieri in [7] and then reformulated by Focardi and Rossi [6], and *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC), proposed in [6] as a strenghtening of BNDC, better suited to deal with dynamic contexts.

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

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

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

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

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

Figure 1: Operational Semantics of CCS Processes

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

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

Thus $P \stackrel{\tau}{\Longrightarrow} P'$ requires at least one τ-transition while $P \stackrel{\hat{\tau}}{\Longrightarrow} P'$ allows for the empty move.

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 $\alpha \in Act$:*

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

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

It is well known that $\approx$ is the largest weak bisimulation and an equivalence relation.

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

We shall distinguish a particular subset of processes:

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

Note that $\mathcal{P}r^{\mathcal{H}}_{\text{syn}}$ is a syntactic notion, which can be defined inductively. We shall later introduce a semantic notion of high process, defined coinductively, which will be more permissive.

The property of *Bisimulation-based Non Deducibility on Compositions* (BNDC) of [7], in its reformulation given by Focardi and Rossi [6], is now defined as follows (remember that $(\nu\mathcal{H})P$ stands for the restriction of P with respect to $\mathcal{H}$):

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 $\Pi \in \mathcal{P}r^{\mathcal{H}}_{\text{syn}}$, $(\nu\mathcal{H})(P \mid \Pi) \approx (\nu\mathcal{H})P$.*

When there is no ambiguity, we shall simply write BNDC instead of $\mathsf{BNDC}_{\mathcal{H}}$. The intuition behind BNDC is that the low observation of a process P should not be affected by any interaction that P could have with a high environment.

Let us point out two typical sources of insecurity :

1. Insecurity may appear when a high name is followed by a low name in process P, as in $P = a_h.\overline{b_\ell}$, 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 $(\nu\mathcal{H})(P \mid \Pi)$. In this example, choosing $\Pi = \overline{a_h}$, one obtains $(\nu\mathcal{H})(P \mid \Pi) \not\approx (\nu\mathcal{H})P$.

2. Insecurity may also appear when a high name is in conflict with a low name, that is, when they occur in different branches of a sum, as in $P = a_h + \overline{b_\ell}$. Indeed, in this case the conflict is "masked" in the process $(\nu\mathcal{H})P$, while it can be solved in favour of a_h in $(\nu\mathcal{H})(P \mid \Pi)$. Indeed, by taking again $\Pi = \overline{a_h}$ 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 $(\nu\mathcal{H})Q$. Note finally that the process:

$$R = a_h.c_h.\overline{b_\ell} + \overline{b_\ell}$$

is insecure, because for $\Pi = \overline{a_h}$ we get $(\nu\mathcal{H})(R \mid \Pi) \not\approx (\nu\mathcal{H})R$, since $(\nu\mathcal{H})(R \mid \Pi)$ can do a silent move to the deadlocked state $(\nu\mathcal{H})(c_h.\overline{b_\ell})$, which $(\nu\mathcal{H})R$ cannot match.

In [6], Focardi and Rossi showed that BNDC is not strong enough to deal with dynamic contexts, and proposed a more robust property called *Persistent Bisimulation-based Non Deducibility on Compositions* (PBNDC).

To define PBNDC, a new transition relation $\stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$ is required, defined as follows for any $\alpha \in Act$:

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

The transition relation $\stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}}$ is used to define a notion of bisimulation which allows simulation of high actions by inaction or τ-actions:

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 $\alpha \in Act$:*

$$\textit{If } P \stackrel{\alpha}{\longrightarrow} P' \textit{ then there exists } Q' \textit{ such that } Q \stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} Q' \textit{ and } P'\ \mathcal{S}\ Q'.$$

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 $\mathcal{S}$.*

Finally, the property of PBNDC is defined as follows:

Definition 2.6 ($\mathsf{PBNDC}_{\mathcal{H}}$) *Let $P \in \mathcal{P}r$ and $\mathcal{H} \subseteq \mathcal{N}$ be a set of high names. Then P is said to be* persistently secure *with respect to $\mathcal{H}$, $P \in \mathsf{PBNDC}_{\mathcal{H}}$, if $P \approx_{\mathcal{H}} (\nu\mathcal{H})P$.*

Note that, since $(\nu\mathcal{H})P$ cannot perform high actions, the relation $\approx_{\mathcal{H}}$ is essentially used in the definition of PBNDC to allow high moves of P to be simulated by (possibly empty) sequences of τ-moves of $(\nu\mathcal{H})P$.

It was shown in [6] that PBNDC is stronger than BNDC, and that requiring PBNDC for P amounts to requiring BNDC for all reachable states of P (whence the name "persistent"). Intuitively, this is because the quantification over high environments which appears in the definition of BNDC is (implicitly) replaced in the definition of PBNDC by plugging P into a "universal" high environment, capable of persistently matching any high move of P (which in turn corresponds to plugging P in a new arbitrary high environment *at each step*).

All the examples considered above are treated in the same way by BNDC and PBNDC. To illustrate the difference between the two notions, we give an example of a process which is secure but not persistently secure. This example is taken from [6] (with a little variation due to the fact that τ-prefixes are not allowed in our syntax, so we need to encode them with parallel composition and restriction).

Example 2.1 (Secure but not persistently secure process)
The process $P = P_1 + P_2 = a_\ell.b_h.\overline{c_\ell} + a_\ell.(\nu d_\ell)(\overline{d_\ell} \mid d_\ell.\overline{c_\ell} \mid d_\ell)$ *is not persistently secure since its reachable state* $b_h.\overline{c_\ell}$ *is not secure. On the other hand* P *is secure, as it may be seen by choosing* $\Pi = \overline{b_h}$*, which is the only candidate for detecting insecurity, and proving that* $(\nu\mathcal{H})(P \mid \Pi) \approx (\nu\mathcal{H})P$*. Indeed, if either of the two processes chooses to execute the second summand* P_2*, reducing it to* $P_2' = (\nu d_\ell)(\overline{d_\ell} \mid d_\ell.\overline{c_\ell} \mid d_\ell)$*, the other process can reply with exactly the same move, since* P_2' *does not contain high channels and thus the residuals* $(\nu\mathcal{H})(P_2' \mid \Pi)$ *and* $(\nu\mathcal{H})P_2'$ *are trivially equivalent. If on the other hand* $(\nu\mathcal{H})(P \mid \Pi)$ *chooses to execute the first summand* P_1*, reducing it to* $P_1' = b_h.\overline{c_\ell}$*, then* $(\nu\mathcal{H})P$ *can reply by the sequence of moves* $\xrightarrow{a_\ell}\xrightarrow{\tau}$ *of* P_2*, where the* τ *corresponds to the first synchronisation, since the respective residuals* $(\nu\mathcal{H})(P_1' \mid \Pi)$ *and* $(\nu\mathcal{H})(\nu d_\ell)(\mathbf{0} \mid \overline{c_\ell} \mid d_\ell)$ *are both equivalent to* $\overline{c_\ell}$*. Conversely, if* $(\nu\mathcal{H})P$ *chooses to execute the first summand* P_1 *reducing it to* P_1'*, then* $(\nu\mathcal{H})(P \mid \Pi)$ *can reply by the sequence of moves* $\xrightarrow{a_\ell}\xrightarrow{\tau}$ *of* P_2*, where the* τ *corresponds now to the second synchronisation, since the respective residuals* $(\nu\mathcal{H})P_1'$ *and* $(\nu\mathcal{H})(\nu d_\ell)((\mathbf{0} \mid d_\ell.\overline{c_\ell} \mid \mathbf{0}) \mid \Pi)$ *are both equivalent to* $\mathbf{0}$*.*

2.3 A security type system for PBNDC

In this section we present a security type system for CCS, which ensures the property of PBNDC. This type system can be viewed as the reduction to CCS of the security type systems proposed for the π-calculus by Pottier [21] and by Hennessy et al. [10, 11].

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 $\{l, h\}$, with $l \leq h$, to match the partition of the set of channels into $\mathcal{L}$ and $\mathcal{H}$. However all our results about the type system would hold for an arbitrary lattice of security levels.

A *type environment* Γ is a mapping from channels to security levels, together with a partial mapping from process identifiers to security levels (this second part of the environment is needed to type recursive processes). This mapping is extended to prefixes and visible actions by letting $\Gamma(\pi) = \Gamma(subj(\pi))$ and for any $\alpha \neq \tau$, $\Gamma(\alpha) = \Gamma(subj(\alpha))$.

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 for CCS are given in Figure 2. Let us discuss the most interesting ones.

(Sum) 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 $\sigma \leq \sigma_i$. Consider for instance the process $P = a_\ell.\overline{b_\ell} + a_\ell.\overline{c_h}$. Then rule (Sum) yields $\Gamma \vdash_\ell P$ because, assuming Γ is the type environment specified by the subscripts, we have $\Gamma(\pi_1) = \Gamma(\pi_2) = \ell$, as well as $\Gamma \vdash_\ell P_1$ and $\Gamma \vdash_\ell P_2$, where the latter judgement is deduced using rule (Sub) from $\Gamma \vdash_h P_2$. Note that, as expected, processes $a_h.\overline{b_\ell}$ and $a_h + \overline{b_\ell}$ are not typable. On the other hand, it should be pointed out that the process $a_h.\overline{b_\ell} + \overline{b_\ell}$, which was argued to be secure at page 8, is not typable either. Hence (Sum) is stricter than we would wish. In order

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

$$\text{(Res)}\quad \frac{\Gamma,\, b:\theta \vdash_\sigma P}{\Gamma \vdash_\sigma (\nu b)P} \qquad \text{(Sub)}\quad \frac{\Gamma \vdash_\sigma P \qquad \sigma' \leq \sigma}{\Gamma \vdash_{\sigma'} P}$$

$$\text{(Rec}_1\text{)}\quad \frac{\Gamma(A) = \sigma}{\Gamma \vdash_\sigma A(\tilde{e})} \qquad \text{(Rec}_2\text{)}\quad \frac{\Gamma,\, A:\sigma \vdash_\sigma P}{\Gamma \vdash_\sigma (\mathtt{rec}\, A(\tilde{x})\,.\,P)(\tilde{e})}$$

Figure 2: Type system for CCS

to make process $a_h.\,\overline{b_\ell} + \overline{b_\ell}$ typable, we could replace rule (Sum) by a more permissive 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:

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

Note that rule (Sum-Lax) makes use of the semantic equivalence $\approx_{\mathcal{H}}$, and hence is not completely static. We shall come back to this rule later in the paper. For the time being, we shall stick to the more classical type system of Figure 2.
Note finally that it would not have been clear how to deal with the full CCS language, that is, with arbitrary sums rather than just guarded sums. Consider for instance the process $a_\ell + (b_h \mid c_\ell)$. The second summand has type ℓ by rule (Par). Then, if rule (Sum) were generalised to full CCS by requiring each summand to have the same type, this process, which is clearly insecure, would be typable with type ℓ.

(Rec$_2$) This rule states that, in order to give type σ to the body P of a recursive process $\mathtt{rec}A(\tilde{x})\,.\,P$, and hence to the whole process, it is necessary to assume that the identifier A itself has type σ. Rule (Rec$_1$) is used to type instantiated process identifiers and to prove the hypothesis of rule (Rec$_2$).

Some simple properties of this type system can immediately be established.

Lemma 2.2 (Substitution preserves types) *Let $P \in \mathcal{P}r$. Then:*

1. *If $\Gamma \vdash_\sigma P$, then for any $v \in Val$ and $x \in fv(P)$, $\Gamma \vdash_\sigma P\{v/x\}$.*
2. *If $\Gamma, A : \sigma \vdash_\sigma P$ and $Q \in \mathcal{P}r$ is such that $\Gamma \vdash_\sigma Q$, then $\Gamma \vdash_\sigma P\{Q/A\}$.*

Theorem 2.3 (Subject reduction)
For any $P \in \mathcal{P}r$, if $\Gamma \vdash_\sigma P$ and $P \xrightarrow{\alpha} P'$ then $\Gamma \vdash_\sigma P'$.

Proof By induction on the inference of $\Gamma \vdash_\sigma P$. The proof is reported in the Appendix. □

We proceed now to establish the soundness of the above type system for the property of PBNDCon CCS. To this end, we need a number of preliminary results. The first one corresponds to the property usually called *confinement* in imperative languages [32]. It states that types have the intended meaning.

Lemma 2.4 (Confinement)
Let $P \in \mathcal{P}r$ and $\Gamma \vdash_\sigma P$. If $P \xrightarrow{\alpha} P'$ and $\alpha \neq \tau$ then $\Gamma(\alpha) \geq \sigma$.

Proof By induction on the proof of $\Gamma \vdash_\sigma P$. The proof may be found in the Appendix. □

There is a set of processes for which the security property is particularly easy to establish because of their inability to perform low actions. These are the *semantically high processes*, which can be defined either relatively to a partition $\{\mathcal{L}, \mathcal{H}\}$ of the set of channels $\mathcal{N}$, or with respect to a type environment Γ.

Definition 2.7 (Semantically high processes w.r.t. $\mathcal{H}$)
Let $\mathcal{H} \subseteq \mathcal{N}$. Then the set of semantically high processes with respect to $\mathcal{H}$, denoted $\mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$, is the largest set such that $P \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$ implies:

$$\textit{For any } \alpha \in Act, \textit{ if } P \xrightarrow{\alpha} P' \textit{ then } \alpha = \tau \textit{ or } subj(\alpha) \in \mathcal{H}, \textit{ and } P' \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}.$$

Definition 2.8 (Semantically high processes w.r.t. Γ)
Let Γ be a type environment. The set of semantically high programs with respect to Γ, denoted $\mathcal{P}r^{\Gamma}_{\mathtt{sem}}$, is the largest set such that $P \in \mathcal{P}r^{\Gamma}_{\mathtt{sem}}$ implies:

$$\textit{For any } \alpha \in Act, \textit{ if } P \xrightarrow{\alpha} P' \textit{ then } \alpha = \tau \textit{ or } \Gamma(\alpha) = h, \textit{ and } P' \in \mathcal{P}r^{\Gamma}_{\mathtt{sem}}.$$

Fact 2.5 *Let $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$. Then $\mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}} = \mathcal{P}r^{\Gamma}_{\mathtt{sem}}$.*

Lemma 2.6 *Let $P \in \mathcal{P}r$ and $\Gamma \vdash_h P$. Then $P \in \mathcal{P}r^{\Gamma}_{\mathtt{sem}}$.*

Proof By Lemma 2.4 and Theorem 2.3. □

Lemma 2.7 ($\approx_{\mathcal{H}}$ – equivalence of semantically high processes)
Let $P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$. *Then* $P \approx_{\mathcal{H}} Q$.

Proof We show that $\mathcal{S} = \{\, (P,Q) : P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}} \,\}$ is a weak bisimulation up to high. Let $P \xrightarrow{\alpha} P'$. Since $P \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$, we know that $P' \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$ and either $\alpha = \tau$ or $subj(\alpha) \in \mathcal{H}$. In both cases we can choose $Q \xrightarrow{\tau}{}^{*} Q$ as the matching move. □

Lemma 2.8 (Persistent security of semantically high processes)
Let $P \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$. *Then* $P \approx_{\mathcal{H}} (\nu\mathcal{H})P$.

Proof Since $P \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$ implies $(\nu\mathcal{H})P \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$, the result follows by Lemma 2.7. □

We prove now an important property of typable programs, which will be the key for our soundness proof.

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* $P \approx_{\mathcal{H}} P'$.

Proof (*Outline*) Let $\mathcal{S}$ be the binary relation on $\mathcal{P}r$ defined inductively as follows: $(P,Q) \in \mathcal{S}$ if and only if there exist Γ and σ such that $\Gamma \vdash_\sigma P$, $\Gamma \vdash_\sigma Q$ and, letting $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$, one of the following holds:

1. $P \approx_{\mathcal{H}} Q$
2. $P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$
3. $P \xrightarrow{\alpha} Q$ or $Q \xrightarrow{\alpha} P$ for some α such that $\Gamma(\alpha) = h$
4. $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$ and $(P_i, Q_i) \in \mathcal{S}$ for $i = 1, 2$
5. There exists R such that $\Gamma \vdash_\sigma R$, $(P,R) \in \mathcal{S}$ and $(R,Q) \in \mathcal{S}$
6. $P = (\nu b)R$, $Q = (\nu b)S$ and there exists θ such that $\Gamma \cup \{(b,\theta)\} \vdash_\sigma R$, $\Gamma \cup \{(b,\theta)\} \vdash_\sigma S$, and $(R,S) \in \mathcal{S}$
7. $P = (\mathtt{rec}\, A(\tilde{x})\,.\,R)(\tilde{e})$ and $(R\{\tilde{v}/\tilde{x}\}\{\,(\mathtt{rec}\, A(\tilde{x})\,.\,R)\,/\,A\,\},\, Q) \in \mathcal{S}$, with $\tilde{v} = val(\tilde{e})$

It can be shown that $\mathcal{S}$ is a weak bisimulation up to high, by induction on the definition of $\mathcal{S}$. We have to show that for any pair $(P,Q) \in \mathcal{S}$, processes P and Q can do the same weak transitions "up to high" (for the given $\mathcal{H}$), leading to derivatives which are again related by $\mathcal{S}$. The detailed proof is given in the Appendix. □

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 $P \mid R \approx_{\mathcal{H}} Q \mid R$.

Proof It follows from the proof of Lemma 2.9, since the relation $\mathcal{S}$ introduced in that proof contains the pair $(P \mid R,\, Q \mid R)$ and is such that $\mathcal{S} \subseteq \approx_{\mathcal{H}}$. □

It should be noted that $\approx_{\mathcal{H}}$ is not preserved by parallel composition on arbitrary programs, as shown by the following 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 $P_1 \mid P_2 \not\approx_{\mathcal{H}} Q_1 \mid Q_2$ because $P_1 \mid P_2$ can perform a τ-action leading to $\mathbf{0}$, which $Q_1 \mid Q_2$ cannot match: indeed, $Q_1 \mid Q_2$ cannot reply by the empty move because this would not discard the low action b_ℓ (nor can it reply by the high action $\overline{a_h}$, since this is not allowed by $\approx_{\mathcal{H}}$). Note that P_2 is not typable, because it does not meet the requirements of rule (Sum). Indeed, P_2 is insecure. Let us point out that, although $\approx_{\mathcal{H}}$ is not compositional, the property of PBNDC itself is compositional, as shown in [6].

We may now prove our main result, namely 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 $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$.

Proof We show that the following relation is a weak bisimulation up to high (then the result will follow since $(P, (\nu\mathcal{H})P\,)$ belongs to this relation):

$$\mathcal{S} = \{\, (P, (\nu\mathcal{H})Q\,) \,:\, \Gamma \vdash_\sigma P,\ \ \Gamma \vdash_\sigma Q, \mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\},\ \ P \approx_{\mathcal{H}} Q\}$$

From $P \approx_{\mathcal{H}} Q$, it follows that P can match the moves of $(\nu\mathcal{H})Q$, since these are a subset of those of Q. Note that subject reduction is used here, since the derivatives must have the same type σ. We consider the other direction.

Let $P \xrightarrow{\alpha} P'$. There are two cases to consider:

- Suppose $subj(\alpha) \in \mathcal{L}$ or $subj(\alpha) = \tau$. From $P \approx_{\mathcal{H}} Q$, it follows that there exists Q' such that $Q \stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} Q'$ and $P' \approx_{\mathcal{H}} Q'$. Then also $(\nu\mathcal{H})Q \stackrel{\widetilde{\alpha}}{\Longrightarrow}_{\mathcal{H}} (\nu\mathcal{H})Q'$. Since $\Gamma \vdash_\sigma P'$ and $\Gamma \vdash_\sigma Q'$ by subject reduction, we get $(P', (\nu\mathcal{H})\, Q') \in \mathcal{S}$.
- Suppose now $subj(\alpha) \in \mathcal{H}$. Then $P \approx_{\mathcal{H}} P'$ by Lemma 2.9 and thus also $P' \approx_{\mathcal{H}} Q$. By subject reduction $\Gamma \vdash_\sigma P'$. Then $(P', (\nu\mathcal{H})Q) \in \mathcal{S}$, thus $(\nu\mathcal{H})Q$ can match the move $P \xrightarrow{\alpha} P'$ by the empty move $(\nu\mathcal{H})Q \xrightarrow{\tau}{}^{*} (\nu\mathcal{H})Q$.

□

This concludes, for the time being, our discussion about noninterference and types for CCS. We shall turn now our attention to a language for parallel imperative programs.

3 Translating parallel imperative programs into CCS

Our aim is to establish a correspondence between language-based NI and process-based NI, and between the associated type systems. We focus on the simple parallel imperative language studied by Smith and Volpano in [30]. Since this is essentially the language IMP considered by Winskel in [34], extended with a parallel operator, we shall call it PARIMP.

Several NI properties and related type systems have already been proposed for PARIMP, e.g. [27, 1, 29, 4], inspired from the pioneering work by Volpano, Smith et al. [32, 30, 31]. In the previous section we have studied a type system for CCS, ensuring the property of PBNDC. The next step is to devise a mapping from PARIMP to CCS, which preserves both a chosen NI property and the associated typing. Now, there exists a well known translation of PARIMP into CCS, given by Milner in [18]. In [8], Focardi, Rossi and Sabelfeld showed that a variant of Milner's translation preserves – by mapping it to PBNDC– a time-sensitive notion of NI for IMP, 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[1]. We will show that this property is preserved by a simpler variant of Milner's translation.

3.1 The imperative language PARIMP

In this section we recall the syntax and semantics of the language PARIMP, and we define our property of security, a time-insensitive NI property inspired from [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' [2]. 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 given by:

$$\begin{aligned} C,\, D \quad ::= \quad & \texttt{nil} \mid X := E \mid C\,;\, D \mid (\texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ D) \mid \\ & (\texttt{while}\ E\ \texttt{do}\ C) \mid (C \parallel D) \end{aligned}$$

The operational semantics of the language is given in terms of transitions between configurations $\langle C, s\rangle \to \langle C', s'\rangle$ 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 $\to$, and $\to^*$ for the reflexive and transitive closure of $\to$. The operational rules for configurations are given in Figure 3. These rules are mostly standard, thus we shall not comment about them. Let

[1] Note that PBNDC is itself time-insensitive.

[2] In general, we shall use uppercase letters for syntactic categories of PARIMP, in order to distinguish them from those of CCS.

us just remark that rules (Seq-Op2), (ParL-Op2) and (ParR-Op2) are introduced, as in [3], to allow every terminated configuration to take the form $\langle \mathtt{nil}, s\rangle$[3].

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 $\langle C, s\rangle \to \langle C', s'\rangle$ 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 a partition of the set of variables into a set of *low variables* L and a set of *high variables* H. In examples, we shall 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, on which the definition of security is based.

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. In particular, if the first program modifies the low part of s, then the simulating program must modify the low part of t (and thus perform a proper move), in order to meet the condition $s' =_L t'$. If on the other hand the first program does not touch the low part of s, i.e. $s' =_L s$, then the second program may reply by staying idle. In this sense, the relation $\simeq_L$ is time-insensitive. It is also termination-insensitive, as it relates any two programs which do not modify the low state, as for instance $\mathtt{nil}$ and $(\mathtt{while}\ tt\ \mathtt{do}\ \mathtt{nil})$. 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 "low memory traces", as illustrated by Example 3.1 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 3.1 *The following program, where $\mathtt{loop}\ D \stackrel{\text{def}}{=} (\mathtt{while}\ tt\ \mathtt{do}\ D)$:*

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

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] In contrast, Rules (ParL-Op2) and (ParR-Op2) were not needed in [4], where the syntax was slightly more restrictive, in that the first component of a process $P; Q$ was assumed to be sequential.

$$\text{(Assign-Op)}\quad \frac{}{\langle X := E, s\rangle \to \langle \texttt{nil}, s[s(E)/X]\rangle}$$

$$\text{(Seq-Op1)}\quad \frac{\langle C, s\rangle \to \langle C', s'\rangle}{\langle C; D, s\rangle \to \langle C'; D, s'\rangle}$$

$$\text{(Seq-Op2)}\quad \frac{}{\langle \texttt{nil}; D, s\rangle \to \langle D, s\rangle}$$

$$\text{(Cond-Op1)}\quad \frac{s(E) = tt}{\langle \texttt{if } E \texttt{ then } C \texttt{ else } D, s\rangle \to \langle C, s\rangle}$$

$$\text{(Cond-Op2)}\quad \frac{s(E) \neq tt}{\langle \texttt{if } E \texttt{ then } C \texttt{ else } D, s\rangle \to \langle D, s\rangle}$$

$$\text{(While-Op1)}\quad \frac{s(E) = tt}{\langle \texttt{while } E \texttt{ do } C, s\rangle \to \langle C; \texttt{while } E \texttt{ do } C, s\rangle}$$

$$\text{(While-Op2)}\quad \frac{s(E) \neq tt}{\langle \texttt{while } E \texttt{ do } C, s\rangle \to \langle \texttt{nil}, s\rangle}$$

$$\text{(ParL-Op1)}\quad \frac{\langle C, s\rangle \to \langle C', s'\rangle}{\langle C \parallel D, s\rangle \to \langle C' \parallel D, s'\rangle}$$

$$\text{(ParL-Op2)}\quad \frac{}{\langle \texttt{nil} \parallel D, s\rangle \to \langle D, s\rangle}$$

$$\text{(ParR-Op1)}\quad \frac{\langle D, s\rangle \to \langle D', s'\rangle}{\langle C \parallel D, s\rangle \to \langle C \parallel D', s'\rangle}$$

$$\text{(ParR-Op2)}\quad \frac{}{\langle C \parallel \texttt{nil}, s\rangle \to \langle C, s\rangle}$$

Figure 3: Operational Semantics of PARIMP

3.2 Milner's translation of PARIMP into CCS

We review now Milner's translation of the language PARIMP into CCS [18]. This translation makes use of two new constructs of CCS, renaming and conditional, which were not present in the language of Section 2. Their operational semantics is well-known and is recalled for completeness in the Appendix.

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

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

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

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

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

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

The channel `res` is used by the auxiliary operator *Into*, defined as follows, where the process P is assumed to be the translation of an expression E:

$$P \; Into\,(x)\; Q \stackrel{\text{def}}{=} (P \mid \mathtt{res(x)}.\,Q)\backslash\mathtt{res}$$

Finally, in order to model sequential composition in CCS, a distinguished channel `done` is introduced, through which processes may signal their termination. Each process $[\![C]\!]$ obtained from a command C will emit an action $\overline{\mathtt{done}}$ upon termination. The idea is that, in the translation of the sequential composition $C\,;\,D$, the termination signal of $[\![C]\!]$ will serve as a trigger for $[\![D]\!]$.

The channel `done` is used by the auxiliary operators *Done*, *Before* and *Par*, which are defined as follows, assuming d, d_1, d_2 to be new names:

$$\begin{aligned}
&Done \stackrel{\text{def}}{=} \overline{\mathtt{done}}.\,\mathbf{0}\\
&C \; Before \; D \stackrel{\text{def}}{=} (C[d/\mathtt{done}] \mid d.\,D)\backslash d\\
&C_1 \; Par \; C_2 \stackrel{\text{def}}{=} ((C_1[d_1/\mathtt{done}] \mid C_2[d_2/\mathtt{done}]) \mid \\
&\qquad\qquad (d_1.\,d_2.Done + d_2.\,d_1.Done))\backslash\{d_1, d_2\}
\end{aligned}$$

The translation of commands is then given by:

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

The translation of a well-formed configuration $\langle C, s\rangle$ is defined as follows:

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

where Acc_s is the *access sort* of state s, namely the set of channels by which processes access variables in the domain of s:

$$\mathit{Acc}_s \stackrel{\text{def}}{=} \{\, \mathit{get}_X, \mathit{put}_X \;\mid\; X \in \texttt{dom}(s)\,\}$$

3.3 Adapting the translation to preserve noninterference

As noted by Milner in [18], the translation just described does not preserve the atomicity of assignment statements. Consider the program: $C = (X := X + 1 \parallel X := X + 1)$. The translation of C is (omitting trailing $\mathbf{0}$'s):

$$\begin{array}{rcl}
[\![C]\!] & = & (\, (\mathit{get}_X(x).\, \overline{\texttt{res}}\langle x+1\rangle \mid \texttt{res}(y).\, \overline{\mathit{put}_X}\langle y\rangle.\, \overline{d_1})\setminus\texttt{res} \\
 & \mid & (\mathit{get}_X(x).\, \overline{\texttt{res}}\langle x+1\rangle \mid \texttt{res}(y).\, \overline{\mathit{put}_X}\langle y\rangle.\, \overline{d_2})\setminus\texttt{res} \\
 & \mid & (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})\,) \setminus \{d_1, d_2\}
\end{array}$$

Here the second get_X action may be executed before the first $\overline{\mathit{put}_X}$ 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 the original program only produces the final value $v_2 = v_0 + 2$ for X, the target process may also produce the final value $v_1 = v_0 + 1$.

It is then easy to see that the translation does not preserve the security of programs. Let $C_L = (X_L := X_L + 1 \parallel X_L := X_L + 1)$ and $D_L = (X_L := X_L + 1\,;\, X_L := X_L + 1)$. Consider now the program $\widehat{C} = (\texttt{if}\ z_H = 0\ \texttt{then}\ C_L\ \texttt{else}\ D_L)$. Clearly $\widehat{C}$ is secure, since each assignment is executed atomically in PARIMP. On the other hand $[\![\widehat{C}]\!]$ is not secure, because, as we just saw, $[\![C_L]\!]$ and $[\![D_L]\!]$ may have different effects on the low memory.

To get around this example it would be enough to introduce a distinct semaphore for each variable, which would prevent parallel writings into that variable. However this solution

would not take care of the next example, where two parallel assignments modify different variables (here we suppose X, Y to be boolean, so that $\overline{X}$ is the complement of X):

$$C' \quad = \quad (X := \overline{Y} \parallel Y := \overline{X})$$

It is easy to see that, whatever the initial values of X and Y, the program C' will always end up with different final values for X and Y. On the other hand, if overlapping of assignments to different variables is allowed in the target language, then the process $[\![C']\!]$ can read the initial values for X and Y in parallel and, in case they are equal, produce equal final values for them. Let now $C'_L = (X_L := \overline{Y_L} \parallel Y_L := \overline{X_L})$, and let D'_L be the (low-trace equivalent) program[4]: $D'_L = (\texttt{if random-bool then } X_L := \overline{Y_L} \texttt{ else } Y_L := \overline{X_L})$. Then the program $\widehat{C'} = (\texttt{if } z_H = 0 \texttt{ then } C'_L \texttt{ else } D'_L)$ is secure, while $[\![\widehat{C'}]\!]$ is not.

To sum up, in order to preserve program security the translation should prevent the ovelapping of (the images of) concurrent assignments. To ensure such mutual exclusion, we introduce a global semaphore for the whole store:

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

The translation of the assignment statement and of well-formed configurations then becomes:

$$\begin{aligned} [\![X := E]\!] \quad &= \quad \overline{\texttt{lock}}.\,[\![E]\!]\; Into\,(x)\;(\overline{put_X}\langle x\rangle.\,\overline{\texttt{unlock}}.\,Done) \\ [\![\langle C, s\rangle]\!] \quad &= \quad (\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \setminus Acc_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\} \end{aligned}$$

We show now that, in order to preserve security, the translation should additionally preserve the *atomicity of expression evaluation*. Consider the couple of programs:

$$\begin{aligned} C_1 \quad = \quad & X_L := 0\,;\, Y_L := 0\,; \\ & (\,X_L := 1\,;\, Y_L := 1 \parallel (\texttt{if } (X_L = 0 \wedge Y_L = 1) \texttt{ then } Z_L := 0 \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else } Z_L := 1)\,) \\ C_2 \quad = \quad & X_L := 0\,;\, Y_L := 0\,; \\ & (\,X_L := 1\,;\, Y_L := 1 \parallel (\texttt{if } \mathit{ff} \texttt{ then } Z_L := 0 \texttt{ else } Z_L := 1)\,) \end{aligned}$$

It is easy to see that C_1 and C_2 are low-bisimilar, since the expression $(X_L = 0 \wedge Y_L = 1)$ is evaluated atomically, yielding *ff* at each point of execution. On the other hand the process $[\![C_1]\!]$ can fetch the value for X_L before the assignment $X_L := 1$ and the value for Y_L after the assignment $Y_L := 1$, in which case the expression evaluates to *tt*. Therefore $[\![C_1]\!]$ is not low-bisimilar to $[\![C_2]\!]$. Hence, if we put C_1 and C_2 in the branches of a high conditional, as in the previous examples, we obtain a program which is secure, but whose translation is insecure. This shows that expression evaluation should not overlap with assignments, and

[4]Here we use a random boolean value generator `random-bool` to simulate nondeterminism. Strictly speaking, this program is not definable in our language.

thus should be protected with the same semaphore *Sem* which is used for assignments. This means that locks must be introduced also in the translation of conditionals and loops.

We examine two different ways of introducing locks in the translation of conditionals and loops, the first being a simplication of that proposed by Focardi, Rossi and Sabelfeld in [8], while the second is a slightly more structured variant of it.

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

$$\begin{array}{lcl}
[\![(\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2)]\!] & = & \overline{\texttt{lock}}.\,[\![E]\!]\,\mathit{Into}(x)\,(\,\texttt{if } x \texttt{ then } \overline{\texttt{unlock}}.\,[\![C_1]\!] \\
 & & \qquad\qquad\qquad\qquad \texttt{else } \overline{\texttt{unlock}}.\,[\![C_2]\!]\,) \\
[\![(\texttt{while } E \texttt{ do } C)]\!] & = & W, \text{ where } W \stackrel{\text{def}}{=} \overline{\texttt{lock}}.\,[\![E]\!]\,\mathit{Into}(x) \\
 & & (\texttt{if } x \texttt{ then } \overline{\texttt{unlock}}.\,[\![C]\!]\,\mathit{Before}\,W \texttt{ else } \overline{\texttt{unlock}}.\,\mathit{Done})
\end{array}$$

The first translation of PARIMP into CCS, based on Solution 1, is summarised in Figure 4. This is essentially a simpler variant of the translation proposed in [8], where in addition special `tick` actions were introduced in the translation of each statement, 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 for the translation, which would not have held otherwise. We shall come back to this point in the discussion at the end of Section 3.4.

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 simply adapted by replacing $[\![E]\!]$ by $[\![E]\!]_{at}$:

$$\begin{array}{lcl}
[\![(\texttt{if } E \texttt{ then } C_1 \texttt{ else } C_2)]\!] & = & [\![E]\!]_{at}\,\mathit{Into}(x)\;\;(\texttt{if } x \texttt{ then } [\![C_1]\!] \texttt{ else } [\![C_2]\!]) \\
[\![(\texttt{while } E \texttt{ do } C)]\!] & = & W, \text{ where } W \stackrel{\text{def}}{=} [\![E]\!]_{at}\,\mathit{Into}(x) \\
 & & (\texttt{if } x \texttt{ then } [\![C]\!]\,\mathit{Before}\,W \texttt{ else } \mathit{Done})
\end{array}$$

The second translation of PARIMP into CCS, based on Solution 2, is given in Figure 5, where for clarity we denote the translation by the symbol $\langle\!\!\langle\ \rangle\!\!\rangle$ instead of $[\![\ ]\!]$. In the rest of the paper, we shall deal only with the first translation, but all our results hold also for the second translation, with some minor variations in the proofs, discussed in the Appendix.

Semaphore:

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

Translation of states:

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

Translation of expressions:

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

Translation of commands:

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

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

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

Access sort of a state s:

$$\mathit{Acc}_s \stackrel{\text{def}}{=} \{\, get_X, \mathit{put}_X \;\mid\; X \in \texttt{dom}(s)\,\}$$

Figure 4: Translation of PARIMP into CCS

Semaphore:

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

Translation of states:

$$\langle\!|s|\!\rangle \quad = \quad Reg_{X_1}(s(X_1)) \mid \cdots \mid Reg_{X_n}(s(X_n)) \qquad \text{if } \texttt{dom}(s) = \{X_1, \ldots, X_n\}$$

Translation of expressions:

$$\langle\!|F(X_1, \ldots, X_n)|\!\rangle \quad = \quad \underbrace{get_{X_1}(x_1).\cdots.get_{X_n}(x_n)}_{getseq_{\tilde{X}}(\tilde{x})}.\,\overline{\texttt{res}}\langle\,\underbrace{f(x_1, \ldots, x_n)}_{f(\tilde{x})}\,\rangle.\,\mathbf{0}$$

Atomic translation of expressions:

$$\langle\!|F(X_1, \ldots, X_n)|\!\rangle_{at} \quad = \quad \overline{\texttt{lock}}.\,getseq_{\tilde{X}}(\tilde{x}).\,\overline{\texttt{res}}\langle f(\tilde{x})\rangle.\,\overline{\texttt{unlock}}.\,\mathbf{0}$$

Translation of commands:

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

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

$$\langle\!|\langle C, s\rangle|\!\rangle = (\,\langle\!|C|\!\rangle \mid \langle\!|s|\!\rangle \mid \mathit{Sem}\,) \setminus \mathit{Acc}_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\}$$

Access sort of a state s:

$$\mathit{Acc}_s \stackrel{\text{def}}{=} \{\, get_X, put_X \;\mid\; X \in \texttt{dom}(s)\,\}$$

Figure 5: Alternative translation of PARIMP into CCS

3.4 The translation preserves security

We show now that the translation described in Figure 4 preserves security. Such a result is 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 the behaviour of its image, we must provide a means to observe the changes performed by $[\![C]\!]$ on $[\![s]\!]$ in CCS[5]. To this end we introduce, as in [25, 8], 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 modified as follows.

The *observable register* $OReg_X$ associated with variable X is defined by:

$$\begin{aligned} OReg_X(v) \stackrel{\text{def}}{=}\ & put_X(x).\, OReg_X(x) \ +\ \overline{get_X}\langle v\rangle.\, OReg_X(v) \ + \\ & \overline{\texttt{lock}}.\,(\, in_X(x).\, \overline{\texttt{unlock}}.\, OReg_X(x) \ +\ \overline{\texttt{unlock}}.\, OReg_X(v)\,) \ + \\ & \overline{\texttt{lock}}.\,(\, \overline{out_X}\langle v\rangle.\, \overline{\texttt{unlock}}.\, OReg_X(v) \ +\ \overline{\texttt{unlock}}.\, OReg_X(v)\,) \end{aligned}$$

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 also that, after committing to communicate with the environment by means of a $\overline{\texttt{lock}}$ action, an observable register can always withdraw its commitment by doing an $\overline{\texttt{unlock}}$ action, and get back to its initial state. This ensures that the pool of observable registers, when put in parallel with the semaphore, is always weakly bisimilar to the image of some state s.

Assume now that each Reg_X is replaced by $OReg_X$ in the translation of Figure 4. Suppose that channels $put_X, get_X, in_X, out_X$ have the same security level as variable X, and that channels $\texttt{lock}, \texttt{unlock}, \texttt{res}$ and $\texttt{done}$ are of high level. Suppose also that renaming is only allowed between names of the same security level.

We may then show that the translation preserves security. Let us first establish the operational correspondence between PARIMP programs and their images in CCS.

Notation: Let Env denote the set $\{in_X, out_X \mid X \in Var\}$. For conciseness, we shall write $(\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \upharpoonright Env$ instead of $(\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \setminus Acc_s \cup \{\texttt{done}, \texttt{lock}, \texttt{unlock}\}$. Assume $Env \subseteq \mathcal{N}$, and let $Act_{Env} \stackrel{\text{def}}{=} \{\alpha \in Act \mid subj(\alpha) \in Env\}$. The actions of Act_{Env} will be the only observable actions of processes obtained as images of configurations.

We now introduce labelled transitions $\xrightarrow{in_X v}$ and $\xrightarrow{\overline{out}_X v}$ for configurations[6], as in [8]:

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

[5]Note that, as it stands, the translation maps any configuration $\langle C, s\rangle$ to an unobservable CCS process.
[6]From now on, we shall use v, v' to range over values in PARIMP as well as in CCS.

Remark 3.2 *By definition we have:*

$$\langle C, s\rangle \xrightarrow{in_X v} \langle C', s'\rangle \quad \Leftrightarrow \quad (\, C' = C \ \wedge \ s' = s[v/X]\,)$$
$$\langle C, s\rangle \xrightarrow{\overline{out_X} v} \langle C', s'\rangle \quad \Leftrightarrow \quad (\, s(X) = v \ \wedge \ C' = C \ \wedge \ s' = s\,)$$

We also extend τ-transitions to configurations by letting:

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

We may now define weak labelled transitions $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$ on configurations, where $\alpha \in Act_{Env} \cup \{\tau\}$, 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 3.3 (Program transitions are preserved by the translation)

Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in Act_{Env} \cup \{\tau\}$. Then:

1. *If $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$, $\alpha \neq \tau$, then there exists P such that $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P = [\![\langle C', s'\rangle]\!]$.*
2. *If $\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle$ then there exists P such that $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P \approx [\![\langle C', s'\rangle]\!]$.*

Proof (*Outline*) The proof of Statement 1. is rather straightforward and does not depend on the command C. The proof of Statement 2. is by induction on the proof of the transition $\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle$. The proof of both statements is given in the Appendix. □

Lemma 3.4 (Process transitions are reflected by the translation)

Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in Act_{Env} \cup \{\tau\}$. Then:

1. *If $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$, $\alpha \neq \tau$, then there exist C', s' such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$.*
2. *If $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P$, then either $P \approx [\![\langle C, s\rangle]\!]$ or there exist C', s' such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\tau}{\Longrightarrow} \langle C', s'\rangle$.*

Proof (*Idea*) The proof is considerably more elaborate than that of Lemma 3.3. Intuitively, we need to decompose the computation $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$ or $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P$ 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 Appendix.

□

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 [8]. 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 $R \subseteq Env$, whose semantics is specified by the rule:

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

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

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 $\stackrel{\alpha}{\longmapsto}_H$ on $ResConf$, for any $\alpha \in Act_{Env} \cup \{\tau\}$:

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

The transition relation $\stackrel{\widetilde{\alpha}}{\longmapsto}_H$ is used to define a notion of (quasi-strong) bisimulation on restricted configurations, which allows simulation of high or silent actions by inaction:

Definition 3.4 (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 $cfg_2 \stackrel{\alpha}{\longmapsto}_H cfg_2'$ 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 $\mathcal{S}$.*

We can immediately establish some simple properties of $\sim_H$:

Property 3.5 *Let $R_H, R'_H \subseteq Env_H$. If $\langle C, s\rangle\backslash R_H \sim_H \langle D, t\rangle\backslash R'_H$, then $s =_L t$.*

Proof By definition of $\sim_H$, $\mathtt{dom}(s) = \mathtt{dom}(t)$. Suppose now $X \in (\mathtt{dom}(s) \cap L)$ and $s(X) = v$. By Remark 3.2 there is a move $\langle C, s\rangle \xrightarrow{\overline{out_X}v} \langle C, s\rangle$, from which we deduce, since $out_X \notin R_H$, $\langle C, s\rangle\backslash R_H \xrightarrow{\overline{out_X}v} \langle C, s\rangle\backslash R_H$. Since $\langle C, s\rangle\backslash R_H \sim_H \langle D, t\rangle\backslash R'_H$, there must exist a matching move $\langle D, t\rangle\backslash R'_H \xrightarrow{\overline{out_X}v} \langle D', t'\rangle\backslash R'_H$. Hence $t(X) = v$ by Remark 3.2. □

Property 3.6 *For any* $R \subseteq Env$, $\langle C, s\rangle \sim_H \langle D, t\rangle$ *implies* $\langle C, s\rangle\backslash R \sim_H \langle D, t\rangle\backslash R$.

The following simple properties of $\sim_H$ will be used to prove Proposition 3.11 below:

Lemma 3.7 *For any variable* $X \in L$ *and value* $V \in Val$*:*
$\langle C, s\rangle \sim_H \langle D, t\rangle\backslash Env_H$ *implies* $\langle C, s\,[V/X]\,\rangle \sim_H \langle D, t\,[V/X]\,\rangle\backslash Env_H$.

Lemma 3.8 *For any variable* $X \in H$ *and values* $V, V' \in Val$*:*
$\langle C, s\rangle \sim_H \langle D, t\rangle\backslash Env_H$ *implies* $\langle C, s\,[V/X]\,\rangle \sim_H \langle D, t\,[V'/X]\,\rangle\backslash Env_H$.

Corollary 3.9 *For any variable* $X \in H$ *and value* $V \in Val$*:*
$\langle C, s\rangle \sim_H \langle D, t\rangle\backslash Env_H$ *implies* $\langle C, s\,[V/X]\,\rangle \sim_H \langle D, t\rangle\backslash Env_H$.

Lemma 3.10 $\langle C, s\rangle \sim_H \langle D, t\rangle\backslash Env_H$ *implies* $\langle C, s\rangle\backslash Env_H \sim_H \langle D, t\rangle\backslash Env_H$.

We may now establish the relation between low bisimilarity on programs and bisimilarity up to high on configurations:

Proposition 3.11 (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* $\langle C, s\rangle\backslash Env_H \sim_H \langle D, s\rangle$.

Proof The proof, independent from the syntax of programs, is given in the Appendix. □

Corollary 3.12 (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.*

We may finally prove the main results of this section:

Theorem 3.13 (Bisimilarity up to high is preserved by the translation)

Let $R, R' \subseteq Env_H$*. Then* $\langle C, s\rangle\backslash R \sim_H \langle D, t\rangle\backslash R'$ *implies* $[\![\langle C, s\rangle\backslash R]\!] \approx_{\mathcal{H}} [\![\langle D, t\rangle\backslash R']\!]$*, where* $\mathcal{H} \stackrel{\text{def}}{=} \{\, get_X, put_X, in_X, out_X \mid X \in H \,\} \cup \{\, \mathtt{lock}, \mathtt{unlock}, \mathtt{res}, \mathtt{done} \,\}$.

Proof The proof, which consists in exhibiting a weak bisimulation up to high containing the pair $(\, [\![\langle C, s\rangle\backslash R]\!], [\![\langle D, t\rangle\backslash R']\!] \,)$, may be found in the Appendix. □

Theorem 3.14 (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, the process* $[\![\langle C, s\rangle]\!]$ *satisfies* $\mathsf{PBNDC}_{\mathcal{H}}$*, where* $\mathcal{H} \stackrel{\text{def}}{=} \{get_X, put_X, in_X, out_X \mid X \in H\} \cup \{\, \mathtt{lock}, \mathtt{unlock}, \mathtt{res}, \mathtt{done} \,\}$.

Proof Let s be a state such that $\langle C, s\rangle$ is well-formed. Since C is L-secure, by Corollary 3.12 we have $\langle C, s\rangle \sim_H \langle C, s\rangle\backslash Env_H$. Then by Theorem 3.13, $[\![\langle C, s\rangle]\!] \approx_{\mathcal{H}} [\![\langle C, s\rangle\backslash Env_H]\!]$. Since $[\![\langle C, s\rangle\backslash Env_H]\!] \stackrel{\text{def}}{=} [\![\langle C, s\rangle]\!]\backslash Env_H$, we conclude that $[\![\langle C, s\rangle]\!] \in \mathsf{PBNDC}_{\mathcal{H}}$. □

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 3.5 (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:*

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

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 3.6 (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 $\alpha \in Act_{Env} \cup \{\tau\}$:*

If $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$ then there exists D' such that $\langle D, t\rangle \overset{\alpha}{\longmapsto} \langle D', t'\rangle$ 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 3.15 (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 3.16 (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, $[\![\langle C, s\rangle]\!] \approx [\![\langle D, s\rangle]\!]$.

Proof The statement is simply obtained from that of Theorem 3.13 by noticing that, if $H = \emptyset$, then $R = R' = Env_H = \emptyset$, $\sim_H = \sim$ and $\mathcal{H} = \{\,\texttt{lock}, \texttt{unlock}, \texttt{res}, \texttt{done}\,\}$. Then, since all actions of $\mathcal{H}$ are restricted in $[\![\langle C, s\rangle]\!]$ and $[\![\langle D, s\rangle]\!]$, we have $[\![\langle C, s\rangle]\!] \approx_{\mathcal{H}} [\![\langle D, s\rangle]\!]$ if and only if $[\![\langle C, s\rangle]\!] \approx [\![\langle D, s\rangle]\!]$. □

Note that the equivalence $\simeq$ is rather intensional. For instance it does not in equate the two programs $\texttt{nil}; P$ and P, unless P does not change the memory. On the other hand we

have $P\,;\mathtt{nil} \simeq P$, as well as $(\mathtt{nil} \parallel P) \simeq P$. 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 precisely designed to capture 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 $\mathtt{nil}; P \cong P$ holds in general. This can be obtained as follows.

Let $\langle C, s\rangle \rightsquigarrow \langle C', s'\rangle$ denote the *administrative* transition relation obtained using the subset of rules (Seq-Op2), (ParL-Op1), (ParL-Op2), (ParR-Op1) and (ParR-Op2) of Figure 3, and $\langle C, s\rangle \rightarrow_c \langle C', s'\rangle$ be the *computing* transition relation obtained without using rules (Seq-Op2), (ParL-Op2) and (ParR-Op2). We may then define an equivalence $\cong$ which always identifies programs whose behaviours differ only for administrative moves.

Definition 3.7 (`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:*

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

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

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 definitions of L-bisimulation and L-security may be weakened in a similar way. We conjecture that all the results proved in this section would easily extend to the `nil`-insensitive versions of L-security and behavioural equivalence.

Finally, we may wonder whether the arrow $\mapsto$ could be replaced by the arrow $\rightarrow^*$ 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 and equivalence notions, allowing one execution step to be simulated by several steps, would be closer than ours 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 3.1 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:

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

Here $[\![C]\!]$ is secure while C is not secure (on the other hand it is `nil`-insensitive secure).

At this point of discussion, it may seem surprising that a full abstraction result could be obtained in [8] 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:

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

Both C' and the program C above are insecure with respect to the time-sensitive security property of [8]. On the other hand, it is easy to see that the encodings of these programs without `tick` actions (i.e. according to the translation of Figure 4), 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.

3.5 The translation does not preserve security types

In this section, we show that the type system for CCS presented in Section 2 is not reflected by our translation. We then sketch a solution to overcome this problem.

Consider the following program, which is typable in the type systems of [30, 27, 29, 4]:

$$C = X_H := X_H + 1\,;\, Y_L := Y_L + 1$$

This program is translated to the following process:

$$\begin{aligned}[\![C]\!] \;=\; (\nu d) \quad & (\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_1)\,(get_{X_H}(x).\,\overline{\texttt{res}_1}\langle x+1\rangle \mid \texttt{res}_1(z_1).\,\overline{put_{X_H}}\langle z_1\rangle.\,\overline{\texttt{unlock}}.\,\overline{d}) \;\mid \\ & d.\,\overline{\texttt{lock}}.\,(\nu\,\texttt{res}_2)\,(get_{Y_L}(y).\,\overline{\texttt{res}_2}\langle y+1\rangle \mid \texttt{res}_2(z_2).\,\overline{put_{Y_L}}\langle z_2\rangle.\,\overline{\texttt{unlock}}.\,\overline{\texttt{done}})\,)\end{aligned}$$

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 $\texttt{lock}, \texttt{unlock}$ 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 main parallel components of $[\![C]\!]$, simulating the two assignments: 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 security levels of channels $\texttt{lock}, \texttt{unlock}$ 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`, from which d is obtained by renaming), which carry no values and are restricted. The idea, borrowed from [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 11 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. Then, by imposing restrictions on the use of loops and conditionals in programs (as proposed e.g. 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 assumed to be high. 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 [8] 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 [8], and (4) we proposed a security type system for PBNDC on CCS which, although failing to reflect a 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, and Frédéric Boussinot for useful comments.

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

[7] 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.

[8] 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.

[9] 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.

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

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

[12] S. Arun Kumar and M. Hennessy. An efficiency preorder for processes. *Acta Informatica* 1(29): 737-760, 1992.

[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. To appear in *ACM TOPLAS*. Extended abstract in *Proceedings of POPL'02*, 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*, pages 358-371, 1996.

[17] N. Kobayashi. Type-based Information Flow Analysis for the Pi-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*, 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 Tech. Report n. 0282, 2003.

[24] 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.

[34] G. Winskel. The formal semantics of programming languages. MIT Press, Cambridge MA, 1993.

Appendix

A Proofs of Theorem 2.3, Lemma 2.4 and Lemma 2.9

Theorem 2.3 [Subject reduction]

For any $P \in \mathcal{P}r$, if $\Gamma \vdash_\sigma P$ and $P \xrightarrow{\alpha} P'$ then $\Gamma \vdash_\sigma P'$.

Proof By induction on the inference of $\Gamma \vdash_\sigma P$. The case of (REC$_1$) need not be considered, since in this case no transition is deducible for P. The full proof is reported in the Appendix.

(SUM) Here $P = \sum_{i \in I} \pi_i.P_i$ and $\Gamma \vdash_\sigma P$ is deduced from the hypothesis: for all i, $\Gamma(\pi_i) = \sigma$ and $\Gamma \vdash_\sigma P_i$. Now, $P \xrightarrow{\alpha} P'$ is deduced using either rule (SUM-OP$_1$) or rule (SUM-OP$_2$). In both cases the transition uses some prefix π_i and leads to either $P' = P_i$, in case π_i is an output prefix, or to $P' = P_i\{v/x\}$ for some $v \in Val$, in case π_i is an input prefix $a(x)$. In the first case, since $\Gamma \vdash_\sigma P_i$ we can immediately conclude. In the second case, we use Lemma 2.2 to obtain $\Gamma \vdash_\sigma P_i\{v/x\}$.

(PAR) Here $P = P_1 \mid P_2$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\sigma P_1$ and $\Gamma \vdash_\sigma P_2$. Then the transition $P \xrightarrow{\alpha} P'$ is inferred using one of the three rules (PAR-OP$_1$), (PAR-OP$_2$) and (PAR-OP$_3$). The first two cases are symmetric, so let us assume that $P \xrightarrow{\alpha} P'$ is deduced from $P_1 \xrightarrow{\alpha} P_1'$ and $P' = P_1' \mid P_2$. Then by induction $\Gamma \vdash_\sigma P_1'$, thus by rule (PAR) also $\Gamma \vdash_\sigma P_1' \mid P_2$. Suppose now $\alpha = \tau$ and $P \xrightarrow{\alpha} P'$ is deduced from $P_1 \xrightarrow{\beta} P_1'$, $P_2 \xrightarrow{\overline{\beta}} P_2'$ for some β such that $subj(\beta) \neq \tau$, and $P' = P_1' \mid P_2'$. By induction $\Gamma \vdash_\sigma P_1'$ and $\Gamma \vdash_\sigma P_2'$, and thus by rule (PAR) also $\Gamma \vdash_\sigma P_1' \mid P_2'$.

(RES) Here $P = (\nu b)R$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma, b : \theta \vdash_\sigma R$. Then $P \xrightarrow{\alpha} P'$ is deduced using rule (RES-OP) from $R \xrightarrow{\alpha} R'$ and $b \neq subj(\alpha)$, and thus $P' = (\nu b)R'$. By induction $\Gamma, b : \theta \vdash_\sigma R'$. Then by rule (RES) also $\Gamma \vdash_\sigma (\nu b)R'$.

(REC$_2$) Here $P = (\texttt{rec}\ A(\tilde{x})\,.\,Q)(\tilde{e})$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma, A : \sigma \vdash_\sigma Q$. Then $P \xrightarrow{\alpha} P'$ is inferred using rule (REC-OP) from $Q\{\tilde{v}/\tilde{x}\}\{\,(\texttt{rec}\ A(\tilde{x})\,.\,Q)\,/\,A\,\} \xrightarrow{\alpha} P'$. By Lemma 2.2, $\Gamma \vdash_\sigma Q\{\tilde{v}/\tilde{x}\}\{\,(\texttt{rec}\ A(\tilde{x})\,.\,Q)\,/\,A\,\}$. Then by induction $\Gamma \vdash_\sigma P'$.

(SUB) Here $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\theta P$ for some $\theta \geq \sigma$. By induction, $P \xrightarrow{\alpha} P'$ implies $\Gamma \vdash_\theta P'$. Then by rule (SUB) also $\Gamma \vdash_\sigma P'$.

□

Lemma 2.4 [Confinement]

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

Proof By induction on the proof of $\Gamma \vdash_\sigma P$. We do not need to consider the case of (REC$_1$).

(SUM) Here $P = \sum_{i\in I} \pi_i.P_i$ and $\Gamma \vdash_\sigma P$ is deduced from the hypotheses $\Gamma(\pi_i) = \sigma$ and $\Gamma \vdash_\sigma P_i$. Now, $P \xrightarrow{\alpha} P'$ is deduced using either rule (SUM-OP$_1$) or rule (SUM-OP$_2$). In both cases, the transition uses some prefix π_i and leads to the corresponding P_i (or $P_i\{v/x\}$ for some $v \in Val$, in case π_i is an input prefix $a(x)$). Then $\Gamma(\alpha) = \Gamma(\pi_i) = \sigma$.

(PAR) Here $P = P_1 \mid P_2$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\sigma P_1$ and $\Gamma \vdash_\sigma P_2$. Since $\alpha \neq \tau$, $P \xrightarrow{\alpha} P'$ is inferred using either rule (PAR-OP$_1$) or rule (PAR-OP$_2$). In both cases we obtain $\Gamma(\alpha) \geq \sigma$ by induction.

(RES) Here $P = (\nu b)Q$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma, b : \theta \vdash_\sigma Q$. Then $P \xrightarrow{\alpha} P'$ is deduced using rule (RES-OP) from $Q \xrightarrow{\alpha} Q'$ and $b \neq \mathit{subj}(\alpha)$. Let $\Gamma' = \Gamma \cup \{(b, \theta)\}$. By induction $\Gamma'(\alpha) \geq \sigma$. Since $\Gamma(\beta) = \Gamma'(\beta)$ whenever $\mathit{subj}(\beta) \neq b$, it follows that $\Gamma(\alpha) = \Gamma'(\alpha) \geq \sigma$.

(REC$_2$) Here $P = (\mathtt{rec}\ A(\tilde{x})\,.\,Q)(\tilde{e})$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma, A : \sigma \vdash_\sigma Q$. Then $P \xrightarrow{\alpha} P'$ is inferred using rule (REC-OP) from $Q\{\tilde{v}/\tilde{x}\}\{\,(\mathtt{rec}\ A(\tilde{x})\,.\,Q)\,/\,A\,\} \xrightarrow{\alpha} P'$. By Lemma 2.2, $\Gamma \vdash_\sigma Q\{\tilde{v}/\tilde{x}\}\{\,(\mathtt{rec}\ A(\tilde{x})\,.\,Q)\,/\,A\,\}$. Then by induction $\Gamma(\alpha) \geq \sigma$.

(SUB) Here $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\theta P$ for some $\theta \geq \sigma$. By induction $\Gamma(\alpha) \geq \theta$, thus by transitivity $\Gamma(\alpha) \geq \sigma$.

□

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 $P \approx_{\mathcal{H}} P'$.

Proof Let $\mathcal{S}$ be the binary relation on $\mathcal{P}r$ defined inductively as follows: $(P, Q) \in \mathcal{S}$ if and only if there exist Γ and σ such that $\Gamma \vdash_\sigma P$, $\Gamma \vdash_\sigma Q$ and, letting $\mathcal{H} = \{\, a \in \mathcal{N} : \Gamma(a) = h \,\}$, one of the following holds:

1. $P \approx_{\mathcal{H}} Q$
2. $P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$
3. $P \xrightarrow{\alpha} Q$ or $Q \xrightarrow{\alpha} P$ for some α such that $\Gamma(\alpha) = h$
4. $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$ and $(P_i, Q_i) \in \mathcal{S}$ for $i = 1, 2$
5. There exists R such that $\Gamma \vdash_\sigma R$, $(P, R) \in \mathcal{S}$ and $(R, Q) \in \mathcal{S}$
6. $P = (\nu b)R$, $Q = (\nu b)S$ and there exists θ such that $\Gamma \cup \{(b, \theta)\} \vdash_\sigma R$, $\Gamma \cup \{(b, \theta)\} \vdash_\sigma S$, and $(R, S) \in \mathcal{S}$
7. $P = (\mathtt{rec}\ A(\tilde{x})\,.\,R)(\tilde{e})$ and $(R\{\tilde{v}/\tilde{x}\}\{\,(\mathtt{rec}\ A(\tilde{x})\,.\,R)\,/\,A\,\},\, Q) \in \mathcal{S}$, with $\tilde{v} = \mathit{val}(\tilde{e})$

We show that $\mathcal{S}$ is a weak bisimulation up to high, by induction on the definition of $\mathcal{S}$. We have to show that for any pair $(P, Q) \in \mathcal{S}$, processes P and Q can do the same weak transitions "up to high" (for the given $\mathcal{H}$), leading to derivatives which are related by $\mathcal{S}$.

1. $P \approx_{\mathcal{H}} Q$. This is the basic case: here the required property holds by definition.

2. $P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$. Then $P \approx_{\mathcal{H}} Q$ by Lemma 2.7, so we fall again in Case 1.

3. It is enough to treat one of the two symmetric cases. Suppose $P \xrightarrow{\alpha} Q$ for some α such that $\Gamma(\alpha) = h$. We use a further induction on the inference of $\Gamma \vdash_\sigma P$.

   (Sum) Here $P = \sum_{i\in I} \pi_i.P_i$ and $\Gamma \vdash_\sigma P$ is deduced from the hypothesis: for all i, $\Gamma(\pi_i) = \sigma$ and $\Gamma \vdash_\sigma P_i$. Now, $P \xrightarrow{\alpha} Q$ is deduced using either rule (SUM-OP$_1$) or rule (SUM-OP$_2$). In both cases, the transition uses some prefix π_i and leads to $Q = P_i$ (or $Q = P_i\{v/x\}$ for some $v \in Val$, in case π_i is an input prefix $a(x)$). Then $\sigma = \Gamma(\pi_i) = \Gamma(\alpha) = h$ and we have $\Gamma \vdash_h P$ and $\Gamma \vdash_h Q$. By Lemma 2.6 it follows that $P, Q \in \mathcal{P}r^{\Gamma}_{\mathtt{sem}}$, and thus, by Fact 2.5, $P, Q \in \mathcal{P}r^{\mathcal{H}}_{\mathtt{sem}}$. Then we fall back into Case 2.

   (Par) Here $P = P_1 \mid P_2$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\sigma P_1$ and $\Gamma \vdash_\sigma P_2$. Since $\alpha \neq \tau$, the transition $P \xrightarrow{\alpha} Q$ is inferred using either rule (PAR-OP$_1$) or rule (PAR-OP$_2$). The two cases are symmetric, so let us assume that $P \xrightarrow{\alpha} Q$ is deduced from $P_1 \xrightarrow{\alpha} P_1'$ and $Q = P_1' \mid P_2$. We want to relate the two processes $P = P_1 \mid P_2$ and $Q = P_1' \mid P_2$. By induction $(P_1, P_1') \in \mathcal{S}$. Since $P_2 \approx_{\mathcal{H}} P_2$, we have $(P_2, P_2) \in \mathcal{S}$ by Clause 1, and thus $(P, Q) = (P_1 \mid P_2,\, P_1' \mid P_2) \in \mathcal{S}$ by Clause 4.

   (Res) Here $P = (\nu b)R$ and $\Gamma \vdash_\sigma P$ is deduced from $\Gamma, b : \theta \vdash_\sigma R$. Then $P \xrightarrow{\alpha} Q$ is deduced using rule (RES-OP) from $R \xrightarrow{\alpha} R'$ and $b \neq \mathit{subj}(\alpha)$, and thus $Q = (\nu b)R'$. By induction $(R, R') \in \mathcal{S}$. Let $\Gamma' = \Gamma \cup \{(b, \theta)\}$. Since $\Gamma' \vdash_\sigma R$, by subject reduction $\Gamma' \vdash_\sigma R'$. Then $(P, Q) = (\,(\nu b)R,\, (\nu b)R'\,) \in \mathcal{S}$ by Clause 6.

   (Rec$_2$) Here $P = (\mathtt{rec}\ A(\tilde{x})\,.\,R)(\tilde{e})$ and $\Gamma \vdash_\sigma P$ is deduced from the hypothesis $\Gamma, A : \sigma \vdash_\sigma R$. Then $P \xrightarrow{\alpha} Q$ is inferred using rule (REC-OP) from $S = R\{\tilde{v}/\tilde{x}\}\{\,(\mathtt{rec}\ A(\tilde{x})\,.\,R)\,/\,A\,\} \xrightarrow{\alpha} Q$. By Lemma 2.2 we know that $\Gamma \vdash_\sigma S$, thus by induction $(S,\, Q) \in \mathcal{S}$ and therefore $(P, Q) \in \mathcal{S}$ by Clause 7.

   (Sub) Here $\Gamma \vdash_\sigma P$ is deduced from $\Gamma \vdash_\theta P$ for some $\theta \geq \sigma$. Since $P \xrightarrow{\alpha} Q$, by subject reduction it follows that $\Gamma \vdash_\theta Q$. Then $(P, Q) \in \mathcal{S}$ by induction.

4. Here $P = P_1 \mid P_2$, $Q = Q_1 \mid Q_2$ and $(P_i, Q_i) \in \mathcal{S}$ for $i = 1, 2$. It is enough to examine the case where P moves first, since the other one is symmetric. So suppose $P \xrightarrow{\beta} P'$. There are two cases to consider, according to whether the move comes from one component (in which case it is deduced using rule (PAR-OP$_1$) or (PAR-OP$_2$)), or from both components (in which case it is deduced using (PAR-OP$_3$)).

(PAR-OP$_1$) In this case $P \xrightarrow{\beta} P'$ is deduced from $P_1 \xrightarrow{\beta} P_1'$, and $P' = P_1' \mid P_2$. Since $(P_1, Q_1) \in \mathcal{S}$, by induction there exists Q_1' such that $Q_1 \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} Q_1'$ and $(P_1', Q_1') \in \mathcal{S}$. Then Q can reply by the transition $Q = Q_1 \mid Q_2 \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} Q_1' \mid Q_2 = Q'$, where $(P_1' \mid P_2,\, Q_1' \mid Q_2) \in \mathcal{S}$ by Clause 4 again.

(PAR-OP$_3$) Here $P \xrightarrow{\tau} P'$ because for some a, v, $P_1 \xrightarrow{av} P_1'$, $P_2 \xrightarrow{\bar{a}v} P_2'$ (or conversely) and $P' = P_1' \mid P_2'$. There are two possible cases:

- If $\Gamma(a) = h$, we have $(P_i', P_i) \in \mathcal{S}$ by Clause 3. By hypothesis $(P_i, Q_i) \in \mathcal{S}$. Hence $(P_i', Q_i) \in \mathcal{S}$ by Clause 5. Therefore Q can reply to the move $P = P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2' = P'$ by the empty move $Q = Q_1 \mid Q_2 \xrightarrow{\tau}{}^* Q_1 \mid Q_2 = Q'$, where $(P', Q') = (P_1' \mid P_2',\, Q_1 \mid Q_2) \in \mathcal{S}$ by Clause 4 again.
- If $\Gamma(a) \neq h$, then by induction $Q_1 \stackrel{\widetilde{av}}{\Longrightarrow}_{\mathcal{H}} Q_1'$ and $Q_2 \stackrel{\widetilde{\bar{a}v}}{\Longrightarrow}_{\mathcal{H}} Q_2'$, with $(P_i', Q_i') \in \mathcal{S}$. Since the actions av and $\bar{a}v$ are not high, these transitions coincide with $Q_1 \stackrel{\widehat{av}}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\widehat{\bar{a}v}}{\Longrightarrow} Q_2'$, and since av and $\bar{a}v$ are visible actions, they further coincide with $Q_1 \stackrel{av}{\Longrightarrow} Q_1'$ and $Q_2 \stackrel{\bar{a}v}{\Longrightarrow} Q_2'$. This means that there exist $\widehat{Q}_i$ and $\widehat{Q}_i'$ such that $Q_1 \xrightarrow{\tau}{}^* \widehat{Q}_1 \xrightarrow{av} \widehat{Q}_1' \xrightarrow{\tau}{}^* Q_1'$ and $Q_2 \xrightarrow{\tau}{}^* \widehat{Q}_2 \xrightarrow{\bar{a}v} \widehat{Q}_2' \xrightarrow{\tau}{}^* Q_2'$. Then Q can reply to the move $P = P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2' = P'$ by the sequence of τ-moves $Q = Q_1 \mid Q_2 \xrightarrow{\tau}{}^* \widehat{Q}_1 \mid \widehat{Q}_2 \xrightarrow{\tau} \widehat{Q}_1' \mid \widehat{Q}_2' \xrightarrow{\tau}{}^* Q_1' \mid Q_2' = Q'$, where $(P', Q') = (P_1' \mid P_2',\, Q_1' \mid Q_2') \in \mathcal{S}$ by Clause 4 again.

5. Here there exists R such that $\Gamma \vdash_\sigma R$, $(P, R) \in \mathcal{S}$ and $(R, Q) \in \mathcal{S}$. Let $P \xrightarrow{\beta} P'$. If $\Gamma(\beta) \neq h$, then by induction $R \stackrel{\widehat{\beta}}{\Longrightarrow} R'$, with $(P', R') \in \mathcal{S}$, and correspondingly $Q \stackrel{\widehat{\beta}}{\Longrightarrow} Q'$, with $(R', Q') \in \mathcal{S}$. Then $(P, Q) \in \mathcal{S}$ by Clause 5 again. If $\Gamma(\beta) = h$, then $(P', P) \in \mathcal{S}$ by Clause 3, and hence $(P', R) \in \mathcal{S}$ by Clause 5, whence $(P', Q) \in \mathcal{S}$, by Clause 5 again. This means that Q can reply to $P \xrightarrow{\beta} P'$ by the empty move $Q \xrightarrow{\tau}{}^* Q$.

6. Here we have $P = (\nu b)R$ and $Q = (\nu b)S$, $(R, S) \in \mathcal{S}$, and there exists θ such that $\Gamma \cup \{(b, \theta)\} \vdash_\sigma R$ and $\Gamma \cup \{(b, \theta)\} \vdash_\sigma S$. In this case a move $P = (\nu b)R \xrightarrow{\beta} (\nu b)R' = P'$ is deduced by rule (RES-OP) from $R \xrightarrow{\beta} R'$ and $b \neq \mathit{subj}(\beta)$. By induction $S \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} S'$, with $(R', S') \in \mathcal{S}$. By subject reduction we have $\Gamma \cup \{(b, \theta)\} \vdash_\sigma R'$ and $\Gamma \cup \{(b, \theta)\} \vdash_\sigma S'$. Then $Q = (\nu b)S \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} (\nu b)S'$, where $((\nu b)R', (\nu b)S') \in \mathcal{S}$ by Clause 6 again.

7. Here $P = (\mathtt{rec}\, A(\tilde{x})\,.\,R)(\tilde{e})$, $(R\{val(\tilde{e})/\tilde{x}\}\{\,(\mathtt{rec}\, A(\tilde{x})\,.\,R)\,/\,A\,\},\, Q) \in \mathcal{S}$, and $P \xrightarrow{\beta} P'$ is inferred by (REC-OP) from $R\{val(\tilde{e})/\tilde{x}\}\{\,(\mathtt{rec}\, A(\tilde{x})\,.\,R)\,/\,A\,\} \xrightarrow{\beta} P'$. By induction $Q \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} Q'$, with $(P', Q') \in \mathcal{S}$. Hence $Q \stackrel{\widetilde{\beta}}{\Longrightarrow}_{\mathcal{H}} Q'$ is the required matching move. □

B Semantics and typing of renaming and conditional

The operational rule for the CCS renaming operator $f[P]$, where f is a finite function from $\mathcal{N}$ to $\mathcal{N}$, extended to actions by letting $f(av) = f(a)v$, $f(\overline{a}v) = \overline{f(a)}v$ and $f(\tau) = \tau$, is:

$$\text{(REN-OP)} \quad \frac{P \xrightarrow{\alpha} P'}{f[P] \xrightarrow{f(\alpha)} f[P']}$$

The typing rule for renaming is (remembering that f is assumed to preserve the security level of channels):

$$(\textsc{Ren}) \quad \frac{\Gamma \vdash_\sigma P}{\Gamma \vdash_\sigma f[P]}$$

The operational rules for the CCS conditional operator are:

$$\text{(COND-OP}_1\text{)} \quad \frac{val(e) = tt}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q) \xrightarrow{\tau} P}$$

$$\text{(COND-OP}_2\text{)} \quad \frac{val(e) = ff}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q) \xrightarrow{\tau} Q}$$

Alternatively, we could have adopted the following rules, as in [18]:

$$\text{(COND-OP}'_1\text{)} \quad \frac{val(e) = tt, \quad P \xrightarrow{\alpha} P'}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q) \xrightarrow{\alpha} P'}$$

$$\text{(COND-OP}'_2\text{)} \quad \frac{val(e) = ff, \quad Q \xrightarrow{\alpha} Q'}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q) \xrightarrow{\alpha} Q'}$$

We choose to use rules (COND-OP$_1$) and (COND-OP$_2$) instead, since they allow for a more precise operational correspondence between PARIMP programs and their CCS images.

The typing rule for conditionals is:

$$(\textsc{Cond}) \qquad \frac{\Gamma \vdash_\sigma P \qquad \Gamma \vdash_\sigma Q}{\Gamma \vdash_\sigma (\texttt{if } e \texttt{ then } P \texttt{ else } Q)}$$

Rule (Cond) may look excessively simple, as it disregards the tested expression. However, since the type system does not allow a high action to be prefixed to a process containing low actions, this rule will be sufficient to exclude an insecure process like:

$$a_H(x).\,(\texttt{if } x = 0 \texttt{ then } \overline{b_\ell}\langle 0\rangle \texttt{ else } \overline{b_\ell}\langle 1\rangle)$$

as well as the process P discussed in the Introduction, which contains an indirect flow.

C Preliminary results about the translation

We recall here some definitions and results from [18].

Definition C.1 (Well-termination)

A process P is well-terminating *if for every derivative P' of P:*

1. *It is not possible that $P' \xrightarrow{\texttt{done}}$*
2. *If $P' \xrightarrow{\overline{\texttt{done}}}$ then $P' \approx \overline{\texttt{done}}.\,\mathbf{0}$*

The following results from [18] also hold for our translation.

Proposition C.1 (Well-termination is preserved by transitions and by $\approx$)

1. *If P is well-terminating and $P \xrightarrow{\alpha} P'$, then P' is well-terminating*
2. *If P is well-terminating and $P \approx Q$, then Q is well-terminating*

Proposition C.2 (The image of a program is well-terminating)

If C is a PARIMP program, then its CCS image $[\![C]\!]$ is well-terminating.

Proposition C.3 (Simple properties of *Before* and *Par*)

For every process P, the following properties hold:

1. *Done Before $P \approx P$*
2. *Done Par $P \approx P$*

Moreover, if P is well-terminating then:

3. *P Before Done $\approx P$*

D Proofs of Lemmas 3.3 and 3.4.

We start by introducing some notation. Given an observable pool of registers $[\![s]\!]$, where $\mathtt{dom}(s) = \{X_1, \ldots, X_n\}$, the possible states of each register $OReg_{X_i}$, as defined at page 3.4:

$$\begin{aligned} OReg_{X_i}(v) \stackrel{\text{def}}{=}\ & put_{X_i}(x).\, OReg_{X_i}(x) \ +\ \overline{get_{X_i}}\langle v\rangle.\, OReg_{X_i}(v) \ + \\ & \overline{\mathtt{lock}}.\,(\, in_{X_i}(x).\, \overline{\mathtt{unlock}}.\, OReg_{X_i}(x) \ +\ \overline{\mathtt{unlock}}.\, OReg_{X_i}(v)\,) \ + \\ & \overline{\mathtt{lock}}.\,(\, \overline{out_{X_i}}\langle v\rangle.\, \overline{\mathtt{unlock}}.\, OReg_{X_i}(v) \ +\ \overline{\mathtt{unlock}}.\, OReg_{X_i}(v)\,) \end{aligned}$$

will be denoted by:

1. $OReg_{X_i}(v)$ (the register associated with X_i, containing value v)
2. $OReg^{in}_{X_i}(v) \stackrel{\text{def}}{=} in_{X_i}(x).\, \overline{\mathtt{unlock}}.\, OReg_{X_i}(x) \ +\ \overline{\mathtt{unlock}}.\, OReg_{X_i}(v)$
3. $OReg^{lock}_{X_i}(v) \stackrel{\text{def}}{=} \overline{\mathtt{unlock}}.\, OReg_{X_i}(v)$
4. $OReg^{out}_{X_i}(v) \stackrel{\text{def}}{=} \overline{out_{X_i}}\langle v\rangle.\, \overline{\mathtt{unlock}}.\, OReg_{X_i}(v) \ +\ \overline{\mathtt{unlock}}.\, OReg_{X_i}(v)$

Correspondingly, the states of the pool of observable registers will be denoted by:

1. $RP(\tilde{v}) \stackrel{\text{def}}{=} OReg_{X_1}(v_1) \mid \cdots \mid OReg_{X_n}(v_n) = [\![s']\!]$, where $\mathtt{dom}(s') = \mathtt{dom}(s)$
2. $RP^{in}_i(\tilde{v}) \stackrel{\text{def}}{=} OReg_{X_1}(v_1) \mid \cdots \mid OReg^{in}_{X_i}(v_i) \mid \cdots \mid OReg_{X_n}(v_n)$
3. $RP^{lock}_i(\tilde{v}) \stackrel{\text{def}}{=} OReg_{X_1}(v_1) \mid \cdots \mid OReg^{lock}_{X_i}(v_i) \mid \cdots \mid OReg_{X_n}(v_n)$
4. $RP^{out}_i(\tilde{v}) \stackrel{\text{def}}{=} OReg_{X_1}(v_1) \mid \cdots \mid OReg^{out}_{X_i}(v_i) \mid \cdots \mid OReg_{X_n}(v_n)$

Let *RP-state* range over $\{\, RP(\tilde{v}), RP^{in}_i(\tilde{v}), RP^{lock}_i(\tilde{v}), RP^{out}_i(\tilde{v}), RP^{out'}_i(\tilde{v}) \mid \tilde{v} = \langle v_1, \ldots, v_n\rangle,\ v_i \in Val\}$. Let also $Sem' = \mathtt{unlock}.\, Sem$, and *Sem-state* range over $\{Sem,\ Sem'\}$.

Definition D.1 (Image-derivatives and abstract states)

A process P is an image-derivative *if there exists a well-formed configuration $\langle C, s\rangle$ such that $[\![\langle C, s\rangle]\!] \stackrel{\hat{\tau}}{\Longrightarrow} P$. If $[\![\langle C, s\rangle]\!] = (\, [\![C]\!] \mid [\![s]\!] \mid Sem\,) \upharpoonright Env \stackrel{\hat{\tau}}{\Longrightarrow} P$, then P has the form:*

$$P = (\, W \mid \textit{RP-state} \mid \textit{Sem-state}\,) \upharpoonright Env$$

for some register pool state RP-state, semaphore state Sem-state and derivative W of $[\![C]\!]$. The component (RP-state | Sem-state) is the abstract state *of the image-derivative P.*

Definition D.2 *An image-derivative $P = (\, W \mid \textit{RP-state} \mid \textit{Sem-state}\,) \upharpoonright Env$ is said to be* unlocked *if Sem-state $= Sem$, and* locked *if Sem-state $= Sem'$.*

Definition D.3 *Let $\equiv$ be the structural congruence on CCS which allows for the rearrangement of parallel components, the elimination of* $\mathbf{0}$ *parallel components and the localisation of restriction and renaming on the components they affect. Clearly, $\equiv$ preserves strong transitions and is stronger than $\approx$.*

Lemma 3.3 [Program transitions are preserved by the translation]

Let $\langle C, s\rangle$ be a well-formed configuration. Then:

1. If $\langle C, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$, $\alpha \neq \tau$, then there exists P such that $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P = [\![\langle C', s'\rangle]\!]$.
2. If $\langle C, s\rangle \xrightarrow{\tau} \langle C', s'\rangle$, there exist P, C'' such that $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P \equiv [\![\langle C'', s'\rangle]\!] \approx [\![\langle C', s'\rangle]\!]$.

Proof The proof of Statement 1. does not depend on the command C. Suppose that $\mathtt{dom}(s) = \{X_1, \ldots, X_n\}$ and $s(X_i) = v_i$. Let $\tilde{v} = \langle v_1, \ldots, v_n\rangle$. There are two possible cases:

- If $\langle C, s\rangle \xrightarrow{in_{X_i} v} \langle C, s[v/X_i]\rangle$ for some v, X_i, then $[\![\langle C, s\rangle]\!] = (\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \upharpoonright Env$ can reply with the sequence of moves, where $\tilde{v}' = \langle v_1, \ldots, \underbrace{v}_{i}, \ldots, v_n\rangle$:

$$\begin{array}{rcl}
(\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \upharpoonright Env & = & (\,[\![C]\!] \mid RP(\tilde{v}) \mid Sem\,) \upharpoonright Env \\
& \xrightarrow{\tau} & (\,[\![C]\!] \mid RP_i^{in}(\tilde{v}) \mid Sem'\,) \upharpoonright Env \\
& \xrightarrow{in_{X_i} v} & (\,[\![C]\!] \mid RP_i^{lock}(\tilde{v}') \mid Sem'\,) \upharpoonright Env \\
& \xrightarrow{\tau} & (\,[\![C]\!] \mid RP(\tilde{v}') \mid Sem\,) \upharpoonright Env \\
& = & [\![\langle C, s[v/X_i]\rangle]\!]
\end{array}$$

- If $\langle C, s\rangle \xrightarrow{\overline{out_{X_i}} v_i} \langle C, s\rangle$ for some X_i, then we have similarly (and more simply):

$$\begin{array}{rcl}
(\,[\![C]\!] \mid [\![s]\!] \mid Sem\,) \upharpoonright Env & = & (\,[\![C]\!] \mid RP(\tilde{v}) \mid Sem\,) \upharpoonright Env \\
& \xrightarrow{\tau} & (\,[\![C]\!] \mid RP_i^{out}(\tilde{v}) \mid Sem'\,) \upharpoonright Env \\
& \xrightarrow{\overline{out_{X_i}} v} & (\,[\![C]\!] \mid RP_i^{lock}(\tilde{v}) \mid Sem'\,) \upharpoonright Env \\
& \xrightarrow{\tau} & (\,[\![C]\!] \mid RP(\tilde{v}) \mid Sem\,) \upharpoonright Env \\
& = & [\![\langle C, s\rangle]\!]
\end{array}$$

This ends the proof of Statement 1.

We prove now Statement 2. by induction on the proof of the transition $\langle C, s\rangle \to \langle C', s'\rangle$.

1. (Assign-Op) In this case the transition is of the form $\langle X := E, s\rangle \to \langle \mathtt{nil}, s[s(E)/X]\rangle$. Correspondingly we have:

$$\begin{array}{rcl}
[\![\langle X := E, s\rangle]\!] & = & (\,\overline{\mathtt{lock}}.\,[\![E]\!]\ \mathit{Into}\,(x)\ (\overline{\mathit{put}_X}\langle x\rangle.\,\overline{\mathtt{unlock}}.\,\mathit{Done}) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \xrightarrow{\tau} & (\,[\![E]\!]\ \mathit{Into}\,(x)\ (\overline{\mathit{put}_X}\langle x\rangle.\,\overline{\mathtt{unlock}}.\,\mathit{Done}) \mid [\![s]\!] \mid \mathtt{unlock}.\,\mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\Longrightarrow} & (\,\mathbf{0} \mid (\overline{\mathit{put}_X}\langle s(E)\rangle.\,\overline{\mathtt{unlock}}.\,\mathit{Done}) \mid [\![s]\!] \mid \mathtt{unlock}.\,\mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \xrightarrow{\tau} & (\,\mathbf{0} \mid (\overline{\mathtt{unlock}}.\,\mathit{Done}) \mid [\![s[s(E)/X]\,]\!] \mid \mathtt{unlock}.\,\mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \xrightarrow{\tau} & (\,\mathbf{0} \mid \mathit{Done} \mid [\![s[s(E)/X]\,]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \;=\; P \\
& \equiv & (\,\mathit{Done} \mid [\![s[s(E)/X]\,]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & [\![\langle \mathtt{nil}, s[s(E)/X]\rangle]\!]
\end{array}$$

2. (Seq-Op1) Here $C = C_1; C_2$ and the transition $\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle$ is deduced from $\langle C_1, s\rangle \to \langle C_1', s'\rangle$. By induction there exist P_1, C_1'' such that $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P_1 \equiv [\![\langle C_1'', s'\rangle]\!] \approx [\![\langle C_1', s'\rangle]\!]$. Then we can deduce:

$$\begin{array}{rcl}
[\![\langle C_1; C_2, s\rangle]\!] & = & (\,([\![C_1]\!]\ \mathit{Before}\,[\![C_2]\!]) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & (\,([\![C_1]\!][d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\Longrightarrow} & (\,(P_1[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d\,) \upharpoonright \mathit{Env} \;=\; P \\
& \equiv & (\,([\![C_1'']\!][d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s']\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \approx & (\,([\![C_1']\!][d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s']\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & (\,([\![C_1']\!]\ \mathit{Before}\,[\![C_2]\!]) \mid [\![s']\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & [\![\langle C_1'; C_2, s'\rangle]\!]
\end{array}$$

3. (Seq-Op2) In this case $C = \mathtt{nil}; C_2$ and $\langle \mathtt{nil}; C_2, s\rangle \to \langle C_2, s\rangle$. Correspondingly:

$$\begin{array}{rcl}
[\![\langle \mathtt{nil}; C_2, s\rangle]\!] & = & (\,([\![\mathtt{nil}]\!]\ \mathit{Before}\,[\![C_2]\!]) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & (\,(\mathit{Done}\,[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \xrightarrow{\tau} & (\,\mathbf{0}\,[d/\mathtt{done}] \mid [\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \;=\; P \\
& \equiv & (\,[\![C_2]\!]) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & [\![\langle C_2, s\rangle]\!]
\end{array}$$

4. (Cond-Op1) Here $C = (\texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2)$ and $\langle \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle \to \langle C_1, s\rangle$ is deduced from $s(E) = tt$. Then we have:

$$\begin{aligned}
[\![\langle \texttt{if}\ E\ \texttt{then}\ C_1\ \texttt{else}\ C_2, s\rangle]\!] &= (\overline{\texttt{lock}}.[\![E]\!]\ \mathit{Into}(x)\ (\texttt{if}\ x\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \texttt{else}\ \overline{\texttt{unlock}}.[\![C_2]\!]) \mid \\
&\qquad [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\Longrightarrow} (\overline{\texttt{res}}\langle tt\rangle.\mathbf{0} \mid \texttt{res}(x).(\texttt{if}\ x\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \texttt{else}\ \overline{\texttt{unlock}}.[\![C_2]\!]) \mid \\
&\qquad [\![s]\!] \mid \texttt{unlock}.\mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\longrightarrow} (\mathbf{0} \mid (\texttt{if}\ tt\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \texttt{else}\ \overline{\texttt{unlock}}.[\![C_2]\!]) \mid \\
&\qquad [\![s]\!] \mid \texttt{unlock}.\mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\longrightarrow} (\mathbf{0} \mid (\overline{\texttt{unlock}}.[\![C_1]\!] \mid [\![s]\!] \mid \texttt{unlock}.\mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\longrightarrow} (\mathbf{0} \mid [\![C_1]\!] \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} = P \\
&\equiv (\,[\![C_1]\!] \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} \\
&= [\![\langle C_1, s\rangle]\!]
\end{aligned}$$

5. (While-Op1) Here $C = (\texttt{while}\ E\ \texttt{do}\ C_1)$ and $\langle \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle \to \langle C_1; \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle$ is deduced from $s(E) = tt$. Then we can deduce, as in the previous case:

$$\begin{aligned}
[\![\langle \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle]\!] &= (W \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} \ \text{ where } W \stackrel{\text{def}}{=} \overline{\texttt{lock}}.[\![E]\!]\ \mathit{Into}(x) \\
&\qquad (\texttt{if}\ x\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \mathit{Before}\ W\ \texttt{else}\ \overline{\texttt{unlock}}.\mathit{Done}) \\
&\stackrel{\tau}{\Longrightarrow} (\mathbf{0} \mid (\texttt{if}\ tt\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \mathit{Before}\ W\ \texttt{else}\ \overline{\texttt{unlock}}.\mathit{Done}) \mid \\
&\qquad [\![s]\!] \mid \texttt{unlock}.\mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\Longrightarrow} (\mathbf{0} \mid [\![C_1]\!]\ \mathit{Before}\ W \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} = P \\
&\equiv (\,[\![C_1]\!]\ \mathit{Before}\ W \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} \\
&= [\![\langle C_1; \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle]\!]
\end{aligned}$$

6. (While-Op2) Here $C = (\texttt{while}\ E\ \texttt{do}\ C_1)$ and $\langle \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle \to \langle \texttt{nil}, s\rangle$ is deduced from $s(E) = \mathit{ff}$. In this case we have:

$$\begin{aligned}
[\![\langle \texttt{while}\ E\ \texttt{do}\ C_1, s\rangle]\!] &\stackrel{\tau}{\Longrightarrow} (\mathbf{0} \mid (\texttt{if}\ \mathit{ff}\ \texttt{then}\ \overline{\texttt{unlock}}.[\![C_1]\!]\ \mathit{Before}\ W\ \texttt{else}\ \overline{\texttt{unlock}}.\mathit{Done}) \mid \\
&\qquad [\![s]\!] \mid \texttt{unlock}.\mathit{Sem}) \upharpoonright \mathit{Env} \\
&\stackrel{\tau}{\Longrightarrow} (\mathbf{0} \mid \mathit{Done} \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} = P \\
&\equiv (\,\mathit{Done} \mid [\![s]\!] \mid \mathit{Sem}) \upharpoonright \mathit{Env} \\
&= [\![\langle \texttt{nil}, s\rangle]\!]
\end{aligned}$$

7. (ParL-Op1) Here $C = (C_1 \parallel C_2)$ and $\langle C_1 \parallel C_2, s\rangle \to \langle C_1' \parallel C_2, s'\rangle$ is deduced from $\langle C_1, s\rangle \to \langle C_1', s'\rangle$. By induction there exist P_1, C_1'' such that $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P_1 \equiv [\![\langle C_1'', s'\rangle]\!] \approx [\![\langle C_1', s'\rangle]\!]$. Then we can deduce:

$$\begin{array}{rcl}
[\![\langle C_1 \parallel C_2\,, s\rangle]\!] & = & (\,(\,[\![C_1]\!]\,\mathit{Par}\,[\![C_2]\!]\,)\mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& = & (\,(\,[\![C_1]\!][d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\backslash\{d_1,d_2\} \mid \\
& & \qquad\qquad [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\Longrightarrow} & (\,P_1[d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\backslash\{d_1,d_2\}\upharpoonright \mathit{Env} \;=\; P \\
& \equiv & (\,(\,[\![C_1'']\!][d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done}))\backslash\{d_1,d_2\} \mid \\
& & \qquad\qquad [\![s']\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& \approx & (\,(\,[\![C_1']\!][d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done}))\backslash\{d_1,d_2\} \mid \\
& & \qquad\qquad [\![s']\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& = & [\![\langle C_1' \parallel C_2, s'\rangle]\!]
\end{array}$$

8. (ParL-Op2) Here $C = (\mathtt{nil} \parallel C_2)$ and the transition is $\langle \mathtt{nil} \parallel C_2, s\rangle \to \langle C_2, s\rangle$. Then we have, using Proposition C.3 to replace $[\![C_2]\!]$ *Before Done* by $[\![C_2]\!]$ in the last line:

$$\begin{array}{rcl}
[\![\langle \mathtt{nil} \parallel C_2, s\rangle]\!] & = & (\,(\,[\![\mathtt{nil}]\!]\,\mathit{Par}\,[\![C_2]\!]\,)\mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& = & (\,(\,\mathit{Done}\,[d_1/\mathtt{done}] \mid [\![C_2]\!]\,[d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\backslash\{d_1,d_2\} \mid \\
& & \qquad\qquad [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\longrightarrow} & (\,(\,\mathbf{0}\,[d_1/\mathtt{done}] \mid [\![C_2]\!]\,[d_2/\mathtt{done}] \mid d_2.\mathit{Done})\backslash\{d_1,d_2\} \mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \;=\; P \\
& \equiv & (\,(\,[\![C_2]\!]\,[d_2/\mathtt{done}] \mid d_2.\mathit{Done})\backslash d_2 \mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& = & (\,(\,[\![C_2]\!]\,\mathit{Before}\,\mathit{Done}\,)\mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \\
& \approx & (\,[\![C_2]\!] \mid [\![s]\!] \mid \mathit{Sem}\,)\upharpoonright \mathit{Env} \;=\; [\![\langle C_2, s\rangle]\!]
\end{array}$$

□

Remark D.1

By inspecting the proof, one may observe that, because of Case 8. (use of Proposition C.3), Statement 2 *of Lemma 3.3* cannot be replaced *by the stronger property:*

$$\langle C, s\rangle \stackrel{\tau}{\longrightarrow} \langle C', s'\rangle \quad \mathit{implies} \quad \exists\, P.\ [\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P \equiv [\![\langle C', s'\rangle]\!]$$

To prove the next result, which relates the computations of a process $[\![\langle C, s\rangle]\!]$ with those of its source configuration $\langle C, s\rangle$, we need to be able to decompose each computation of $[\![\langle C, s\rangle]\!]$ into a sequence of subcomputations corresponding, as closely as possible, to the simulation of a single step of $\langle C, s\rangle$. We start by classifying the possible steps performed by a configuration $\langle C, s\rangle$:

Definition D.4 (Administrative and computing steps)

A configuration $\langle C, s\rangle$ may perform two kinds of steps:

1. administrative steps*, whose effect is to eliminate a* `nil` *subterm: these are inferred using one of the rules* (Seq-Op2)*,* (ParL-Op2)*,* (ParR-Op2) *of Figure 3;*
2. computing steps*, which involve an access to the state: these are further divided into* internal accesses*, inferred by one of the remaining rules of Figure 3, and* external accesses*, derived by one of the rules* (In-Op) *or* (Out-Op) *at page 24.*

The computations that simulate a configuration's step, whose description has been given in the proof of Lemma 3.3, can now be classified correspondingly:

Definition D.5 (Relay moves and transactions)

The steps of a configuration $\langle C, s\rangle$ are simulated in $[\![\langle C, s\rangle]\!]$ as follows:

1. *an administrative step is simulated by a* relay move*, defined as a pure synchronisation between the terminating action of some parallel component and a starting action of another component.*
2. *an internal access is simulated by an* internal transaction*, a minimal sequence of τ-moves which starts by taking the semaphore and ends by releasing it, accessing the state in between.*
3. *an* external access *is simulated by an* external transaction*, a minimal sequence of moves of the form $\xrightarrow{\tau}\xrightarrow{\alpha}\xrightarrow{\tau}$, which starts by taking the semaphore and ends by releasing it, accessing the state in between by means of action α.*

Remark D.2 *In a computation of $[\![\langle C, s\rangle]\!]$, two transactions can never be interleaved, since each of them starts by taking the semaphore and ends by releasing it. On the other hand a transaction may be interleaved with relay moves from some parallel component. Relay moves do not act on the abstract state (as defined at page 41). External transactions act only on the abstract state, and internal transactions act on both $[\![C]\!]$ and the abstract state.*

Due to Remark D.2, we cannot simply decompose a computation of $[\![\langle C, s\rangle]\!]$ into a sequence of transactions and relay moves. We are forced to consider transactions modulo the inclusion of relay moves. To this end we introduce the notion of *micro-computation*. Intuitively, a micro-computation is either a relay move between two unlocked states, or a computation which lies within the simulation of *a single computing step* of the source configuration, i.e. a partial or complete transaction, possibly interspersed with relay moves.

The notion of micro-computation is formalised by means of a new transition relation $P \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P'$, where P is assumed to be an unlocked image-derivative.

Definition D.6 (Micro-computations and micro-derivatives)

Let P be an unlocked image-derivative. A complete micro-computation *of P is either a move $P \stackrel{\tau}{\longrightarrow} P'$, where P' is unlocked, or, for some $\alpha \in Act_{Env} \cup \{\tau\}$, a computation of the form:*

$$P = P_0 \stackrel{\tau}{\longrightarrow} P_1 \cdots P_k \stackrel{\alpha}{\longrightarrow} P_{k+1} \cdots P_{n-1} \stackrel{\tau}{\longrightarrow} P_n = P'$$

where $n > 2,\ 0 < k < n-1$, processes $P_1, \ldots, P_{n-1}$ are locked and process P_n is unlocked.

A micro-computation *is a non empty prefix of a complete micro-computation.*

A micro-computation can be specified directly by means of the transition relation $P \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P'$, defined on image-derivatives as follows. For any $\alpha \in Act_{Env} \cup \{\tau\}$ and image-derivative P:

$$\begin{aligned} P \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P' \quad \Leftrightarrow_{\mathrm{def}} \quad & (\,\alpha = \tau \ \wedge\ P \stackrel{\alpha}{\longrightarrow} P' \ \wedge\ P, P' \textit{ unlocked}\,) \\ \textit{or} \quad & (\,\alpha \in Act_{Env} \cup \{\tau\} \ \wedge\ \exists\, n > 1\,.\,\exists\, k\,.\,0 \le k < n\,.\,\exists\, P_0, \ldots, P_n\,. \\ & \quad P \ =\ P_0 \stackrel{\tau}{\longrightarrow} P_1 \cdots P_k \stackrel{\alpha}{\longrightarrow} P_{k+1} \cdots P_{n-1} \stackrel{\tau}{\longrightarrow} P_n = P' \\ & \quad P_0 \ \textit{unlocked} \ \wedge\ P_1, \ldots, P_{n-1} \ \textit{locked}\,) \end{aligned}$$

If $P \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P'$ then P' is said to be a micro-derivative *of P.*

By definition $P \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P'$ if and only if the underlying computation is a micro-computation. In a micro-computation, only the initial and possibly the final state are unlocked. Thus a micro-computation covers at most one semaphore cycle. A micro-computation is said to be *locking* if it goes through at least one locked state, *unlocking* otherwise. Note that an unlocking micro-computation is always a relay move. The converse is not true since relay moves can also occur between two locked states. However, if $P = [\![\langle C, s\rangle]\!]$ then $P \stackrel{\tau}{\longrightarrow} P'$ is a relay move if and only if it is unlocking, that is, if and only if P' is unlocked.

To prove the next results, we will need to introduce a new relation $\lesssim$ on processes, which we call *relay expansion preorder*, and which is inspired both by the expansion preorder of [28] and by the efficiency preorder of [12]. The intuition for $P \lesssim Q$ is that Q strongly simulates P[7], while P simulates the relay moves of Q either by a relay move or by inaction, and strongly simulates the remaining transitions of Q. This is clearly an asymmetric relation, lying between strong and weak bisimulation, and refining that of [28]. In the next definition and whenever necessary, we shall distinguish relay moves by using the notation $\stackrel{\tau}{\longrightarrow}_r$. The set of relay moves will be denoted by $Relay \stackrel{\mathrm{def}}{=} \{P \stackrel{\tau}{\longrightarrow}_r P' \mid P, P' \in \mathcal{P}r\}$.

[7] According to the classical notion of strong simulation, see [18].

Definition D.7 (Relay expansion preorder) *A relation $\mathcal{S} \subseteq (\mathcal{P}r \times \mathcal{P}r)$ is a* relay expansion relation *if $P\ \mathcal{S}\ Q$ implies, for any $\alpha \in Act$:*

(i) *If $P \xrightarrow{\alpha} P'$ then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P'\ \mathcal{S}\ Q'$.*

(ii) *If $Q \xrightarrow{\alpha} Q'$ and $Q \xrightarrow{\alpha} Q' \notin Relay$, then there exists P' such that $P \xrightarrow{\alpha} P'$ and $P'\ \mathcal{S}\ Q'$.*

(iii) *If $Q \xrightarrow{\tau}_r Q'$ then either $P\ \mathcal{S}\ Q'$ or there exists P' such that $P \xrightarrow{\tau}_r P'$ and $P'\ \mathcal{S}\ Q'$.*

We set $P \lesssim Q$ if $P\ \mathcal{S}\ Q$ for some relay expansion relation $\mathcal{S}$. The relation $\lesssim$ is called the relay expansion preorder. *We shall also use $Q \gtrsim P$ to stand for $P \lesssim Q$.*

We show now that every relay move of the image $[\![\langle C, s\rangle]\!]$ of a configuration is, up to $\gtrsim$, reflected by an administrative step of $\langle C, s\rangle$.

Lemma D.3 (Relay moves are reflected by administrative steps)

Let $\langle C, s\rangle$ be a well-formed configuration. If $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P$ is a relay move, then $\exists\, C', C''$ such that $\langle C, s\rangle \xrightarrow{\tau} \langle C', s\rangle$ and $P \equiv [\![\langle C'', s\rangle]\!] \gtrsim [\![\langle C', s\rangle]\!]$.

Proof By induction on the structure of C. There are only two cases to consider, namely sequential and parallel composition.

1. $C = C_1; C_2$. We distinguish two cases:

   i) If $C_1 = \mathtt{nil}$, the unique relay move $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P$ is:

   $$\begin{array}{rcl} [\![\langle \mathtt{nil}; C_2, s\rangle]\!] & = & (\,([\![\mathtt{nil}]\!]\ \mathit{Before}\ [\![C_2]\!])\mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \\ & = & (\,(\mathit{Done}\,[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \\ & \xrightarrow{\tau} & (\,(\mathbf{0}\,[d/\mathtt{done}] \mid [\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \;=\; P \\ & \equiv & (\,[\![C_2]\!] \mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \\ & = & [\![\langle C_2, s\rangle]\!] \end{array}$$

   Here $P \equiv [\![\langle C_2, s\rangle]\!]$. Since $\langle \mathtt{nil}; C_2, s\rangle \xrightarrow{\tau} \langle C_2, s\rangle$, we may conclude.

   ii) If $C_1 \neq \mathtt{nil}$, then any relay move $[\![\langle C_1; C_2, s\rangle]\!] \xrightarrow{\tau} P$ is of the form:

   $$\begin{array}{rcl} [\![\langle C_1; C_2, s\rangle]\!] & = & (\,([\![C_1]\!]\ \mathit{Before}\ [\![C_2]\!])\mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \\ & = & (\,([\![C_1]\!][d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env} \\ & \xrightarrow{\tau} & (\,P_1[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d\ \restriction\ \mathit{Env} \end{array}$$

   for some P_1 such that $[\![\langle C_1, s\rangle]\!] \xrightarrow{\tau} P_1$.
   By induction there exist C_1', C_1'' such that $\langle C_1, s\rangle \xrightarrow{\tau} \langle C_1', s\rangle$ and $P_1 \equiv [\![\langle C_1'', s\rangle]\!] \gtrsim [\![\langle C_1', s\rangle]\!]$. From $P_1 \equiv [\![\langle C_1'', s\rangle]\!] = (\,[\![C_1'']\!] \mid [\![s]\!] \mid \mathit{Sem}\,)\ \restriction\ \mathit{Env}$ we may now deduce $((P_1[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d)\ \restriction\ \mathit{Env} \equiv ((C_1''[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem})\ \restriction\ \mathit{Env} \gtrsim ((C_1'[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem})\ \restriction\ \mathit{Env} = [\![\langle C_1'; C_2, s\rangle]\!]$. We may then conclude, since $\langle C_1; C_2, s\rangle \xrightarrow{\tau} \langle C_1'; C_2, s\rangle$.

2. $C = (C_1 \parallel C_2)$. Note that the $[\![C_i]\!]$'s cannot communicate with each other, since they have no complementary actions. Then the relay move $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P$ involves only one of the $[\![C_i]\!]$'s, say $[\![C_1]\!]$. Again, there are two cases to consider:

   i) $C_1 = \texttt{nil}$. In this case we have:

$$\begin{aligned}
[\![\langle \texttt{nil} \parallel C_2\,, s\rangle]\!] &= (\,(\,[\![\texttt{nil}]\!]\; \mathit{Par}\; [\![C_2]\!]\,) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&= (\,(\mathit{Done}\,[d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})\,) \setminus \{d_1, d_2\} \mid \\
&\qquad\qquad [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&\xrightarrow{\tau} (\,(\mathbf{0}\,[d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid\; d_2.\mathit{Done}) \setminus \{d_1, d_2\} \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \;=\; P \\
&\equiv (\,([\![C_2]\!][d_2/\texttt{done}] \mid\; d_2.\mathit{Done}) \setminus d_2 \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&= (\,([\![C_2]\!]\; \mathit{Before}\; \mathit{Done}) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&\gtrsim (\,[\![C_2]\!] \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env}
\end{aligned}$$

   Since $\langle \texttt{nil} \parallel C_2, s\rangle \xrightarrow{\tau} \langle C_2, s\rangle$, we may conclude.

   ii) $C_1 \neq \texttt{nil}$. Then the relay move $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P$ is of the form:

$$\begin{aligned}
[\![\langle C_1 \parallel C_2\,, s\rangle]\!] &= (\,(\,[\![C_1]\!]\; \mathit{Par}\; [\![C_2]\!]\,) \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&= (\,([\![C_1]\!][d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})\,) \setminus \{d_1, d_2\} \mid \\
&\qquad\qquad [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&\xrightarrow{\tau} (\,P_1\,[d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})\,) \setminus \{d_1, d_2\} \upharpoonright \mathit{Env}
\end{aligned}$$

   for some P_1 such that $[\![\langle C_1, s\rangle]\!] \xrightarrow{\tau} P_1$.

   By induction there exist C_1', C_1'' such that $\langle C_1, s\rangle \xrightarrow{\tau} \langle C_1', s\rangle$ and $P_1 \equiv [\![\langle C_1'', s\rangle]\!] \gtrsim [\![\langle C_1', s\rangle]\!]$. From $P_1 \equiv [\![\langle C_1'', s\rangle]\!] = (\,[\![C_1'']\!] \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env}$ we deduce:

$$\begin{aligned}
&(\,P_1\,[d_1/\texttt{done}] \mid [\![C_2]\!]\,[d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})\,) \setminus \{d_1, d_2\} \upharpoonright \mathit{Env} \\
&\equiv (\,[\![C_1'']\!][d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})) \setminus \{d_1, d_2\} \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&\gtrsim (\,[\![C_1']\!][d_1/\texttt{done}] \mid [\![C_2]\!][d_2/\texttt{done}] \mid (d_1.\, d_2.\mathit{Done} + d_2.\, d_1.\mathit{Done})) \setminus \{d_1, d_2\} \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
&= [\![\langle C_1' \parallel C_2, s\rangle]\!].
\end{aligned}$$

   Since $\langle C_1 \parallel C_2, s\rangle \xrightarrow{\tau} \langle C_1' \parallel C_2, s\rangle$, we may conclude.

□

Corollary D.4 (Relay computations are reflected by administrative computations)

If $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P_1 \cdots \xrightarrow{\tau} P_n$ *is a sequence of relay moves, then there exist* $C_1, \ldots, C_n$ *such that* $\langle C, s\rangle \overset{\tau}{\longmapsto} \langle C_1, s\rangle \cdots \overset{\tau}{\longmapsto} \langle C_n, s\rangle$ *is a sequence of administrative moves and for each* i, $1 \leq i \leq n$, $P_i \gtrsim [\![\langle C_i, s\rangle]\!]$.

Proof By induction on n. Let $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P_1 \cdots \xrightarrow{\tau} P_n$ be a sequence of relay moves. By Lemma D.3, there exists C_1 such that $\langle C, s\rangle \xrightarrow{\tau} \langle C_1, s\rangle$ and $P_1 \gtrsim [\![\langle C_1, s\rangle]\!]$. So if $n = 1$ the proof ends here. Suppose now $n > 1$. Since $P_1 \xrightarrow{\tau} P_2 \cdots \xrightarrow{\tau} P_n$ and $\gtrsim$ preserves or erases relay moves, there exist $Q_2, \ldots, Q_n$ such that $[\![\langle C_1, s\rangle]\!] \overset{\tau}{\longmapsto} Q_2 \cdots \overset{\tau}{\longmapsto} Q_n$ and for each $i, 2 \leq i \leq n$, $P_i \gtrsim Q_i$. Since the sequence $[\![\langle C_1, s\rangle]\!] \overset{\tau}{\longmapsto} Q_2 \cdots \overset{\tau}{\longmapsto} Q_n$ has length $k \leq n-1$ (because each of its steps is either a relay move or the empty move), by induction there exist $C_2, \ldots, C_n$ such that $\langle C_1, s\rangle \overset{\tau}{\longmapsto} \langle C_2, s\rangle \cdots \overset{\tau}{\longmapsto} \langle C_n, s\rangle$ is a sequence of administrative moves where for each i, $2 \leq i \leq n$, $Q_i \gtrsim [\![\langle C_i, s\rangle]\!]$. Then $\langle C, s\rangle \xrightarrow{\tau} \langle C_1, s\rangle \overset{\tau}{\longmapsto} \langle C_2, s\rangle \cdots \overset{\tau}{\longmapsto} \langle C_n, s\rangle$ is the required administrative computation, since $P_1 \gtrsim [\![\langle C_1, s\rangle]\!]$ and for each $i, 2 \leq i \leq n$, $P_i \gtrsim Q_i \gtrsim [\![\langle C_i, s\rangle]\!]$.. □

The next lemma states that, in a locking micro-computation, relay moves can always be permuted with transaction moves in such a way that all relay moves are executed before the transaction moves, leading to the same final state as the original micro-computation. We call *transaction prefix* any computation which is the initial part of a transaction. A transaction prefix is said to be *proper* if it does not coincide with the complete transaction.

Lemma D.5 (Permutation of transaction moves and relay moves)

Let $\langle C, s\rangle$ *be a well-formed configuration and* $\alpha \in Act_{Env} \cup \{\tau\}$. *If* $[\![\langle C, s\rangle]\!] \overset{\alpha}{\Longmapsto} P_1 \xrightarrow{\tau} P$ *is a micro-computation where* $[\![\langle C, s\rangle]\!] \overset{\alpha}{\Longmapsto} P_1$ *is a proper transaction prefix and* $P_1 \xrightarrow{\tau} P$ *is a relay move, then there exists* P_2 *such that* $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P_2 \overset{\alpha}{\Longmapsto} P$, *where* $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P_2$ *is a relay move and* $P_2 \overset{\alpha}{\Longmapsto} P$ *is a proper transaction prefix.*

Proof The property is straightforward for external transactions, since external transaction moves and relay moves act in parallel on disjoint parts of an image-derivative (the former act only on the abstract state, while the latter do not touch it). We prove the property for internal transactions, by induction on the structure of C. Note that in the case of assignment, conditional and loop statements, there is no possibility that a relay move follows an internal transaction prefix within a single micro-computation. We examine the remaining cases.

1. $C = C_1; C_2$. Note that if $C_1 = \texttt{nil}$ or C_1 reduces to `nil` by a sequence of administrative moves, then $[\![\langle C, s\rangle]\!]$ cannot immediately engage in an internal transaction. Otherwise, any proper internal transaction prefix $[\![\langle C, s\rangle]\!] \overset{\tau}{\Longmapsto} P_1$ has the form:

$$\begin{array}{rcl}
[\![\langle C_1; C_2, s\rangle]\!] & = & (\,([\![C_1]\!]\, \mathit{Before}\, [\![C_2]\!])\mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & (\,([\![C_1]\!][d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d \mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\Longrightarrow} & (\,(Q_1[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d\,) \upharpoonright \mathit{Env} \;=\; P_1
\end{array}$$

for some Q_1 such that $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q_1$.

Since $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q_1$ is a proper transaction prefix, the synchronisation between $\overline{d}$ and d is not enabled, hence any relay move $P_1 \stackrel{\tau}{\longrightarrow} P$ must come from some relay move $Q_1 \stackrel{\tau}{\longrightarrow} Q$. Thus $P = (\,(Q\,[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d\,) \upharpoonright \mathit{Env}$. By induction there exists Q_2 such that $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\longrightarrow} Q_2 \stackrel{\alpha}{\Longrightarrow} Q$, where $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\longrightarrow} Q_2$ is a relay move and $Q_2 \stackrel{\alpha}{\Longrightarrow} Q$ is a proper transaction prefix. Then we have $[\![\langle C_1; C_2, s\rangle]\!] \stackrel{\tau}{\longrightarrow} P_2 \stackrel{\alpha}{\Longrightarrow} P$, where $P_2 = (\,(Q_2\,[d/\mathtt{done}] \mid d.\,[\![C_2]\!])\backslash d\,) \upharpoonright \mathit{Env}$.

2. $C = (C_1 \parallel C_2)$. If both C_i are $\mathtt{nil}$ or reduce to $\mathtt{nil}$ by a sequence of administrative moves, then $[\![\langle C, s\rangle]\!]$ can only perform relay moves and has no internal transactions. Otherwise, any proper internal transaction prefix $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P_1$ is due to one of the components $[\![C_i]\!]$, say $[\![C_1]\!]$ (note that no communication is possible between the components). Hence $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P_1$ has the form:

$$\begin{array}{rcl}
[\![\langle C_1 \parallel C_2\,, s\rangle]\!] & = & (\,(\,[\![C_1]\!]\, \mathit{Par}\, [\![C_2]\!]\,)\mid [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& = & (\,(\,[\![C_1]\!][d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\,\backslash\{d_1, d_2\} \mid \\
& & \qquad\qquad [\![s]\!] \mid \mathit{Sem}\,) \upharpoonright \mathit{Env} \\
& \stackrel{\tau}{\Longrightarrow} & (\,Q_1\,[d_1/\mathtt{done}] \mid [\![C_2]\!][d_2/\mathtt{done}] \mid (d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})\,)\,\backslash\{d_1, d_2\} \upharpoonright \mathit{Env}
\end{array}$$

for some Q_1 such that $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q_1$. Since $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q_1$ is a proper transaction prefix, the synchronisation between $\overline{d_1}$ and d_1 is not enabled, hence any relay move $P_1 \stackrel{\tau}{\longrightarrow} P$ must come either from some relay move $Q_1 \stackrel{\tau}{\longrightarrow} Q$, in which case we proceed by induction as in Case 1. above, or from some relay move involving only $[\![C_2]\!]$, and possibly also the component $(d_1.\,d_2.\mathit{Done} + d_2.\,d_1.\mathit{Done})$, in case $C_2 = \mathtt{nil}$. In the last two cases, the relay move is independent from the transaction prefix $[\![\langle C_1, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q_1$ and thus can be permuted with it, leading to the same final process P.

□

We show now that every transaction prefix of a process $[\![\langle C, s\rangle]\!]$ is reflected in the source configuration $\langle C, s\rangle$ either by a single step or by inaction.

Lemma D.6 (Transaction prefixes are reflected by the translation)
Let $\langle C, s\rangle$ be a well-formed configuration. Then:

1. *If $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$ is a transaction prefix, $\alpha \in Act_{Env}$, then there exist P', C', s' such that $P \stackrel{\hat{\tau}}{\Longrightarrow} P' \equiv [\![\langle C', s'\rangle]\!]$, $P \approx P'$ and $\langle C, s\rangle \stackrel{\alpha}{\longrightarrow} \langle C', s'\rangle$. Moreover, if $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$ is a complete transaction, then $P = P'$.*

2. *If $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ is a transaction prefix, then either $P \approx [\![\langle C, s\rangle]\!]$ or there exist P', C', s' such that $P \stackrel{\hat{\tau}}{\Longrightarrow} P' \equiv [\![\langle C', s'\rangle]\!]$, $P \approx P'$ and $\langle C, s\rangle \stackrel{\tau}{\longrightarrow} \langle C', s'\rangle$. Moreover, if $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ is a complete transaction, then $P = P'$.*

Proof The proof of Statement 1 is obtained by reversing the proof of Lemma 3.3 and observing that, as soon as the external action of $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$ is performed, all the states of the transaction prefix become weakly bisimilar to the final state of the complete transaction.

The proof of Statement 2, in the case where $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ is an *internal transaction* prefix, is obtained in a similar way. Here, as soon as the locking step of $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ is performed, all states become weakly bisimilar to the final state of the complete transaction. Suppose now that $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ consists of a single step which is the locking step of an *external transaction*. This means that for some $X_i \in \mathtt{dom}(s)$, the register $OReg_{X_i}(v)$, where $v = s(X_i)$, has moved to one of the states $OReg^{in}_{X_i}(v)$ or $OReg^{out}_{X_i}(v)$ (as defined at page 41). Then $P \approx [\![\langle C, s\rangle]\!]$, because P may release the lock and come back to $[\![\langle C, s\rangle]\!]$. □

We are now able to prove that every micro-computation of a process $[\![\langle C, s\rangle]\!]$ is reflected by a possibly empty sequence of steps of $\langle C, s\rangle$, at most one of which is a computing step.

Proposition D.7 (Micro-computations are reflected by the translation)
Let $\langle C, s\rangle$ be a well-formed configuration. Then:

1. *If $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$, $\alpha \in Act_{Env}$, then there exist C', s' such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$. Moreover, if $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$ is a complete micro-computation, then $P \gtrsim [\![\langle C', s'\rangle]\!]$.*

2. *If $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$, then either $P \approx [\![\langle C, s\rangle]\!]$ or there exist C', s' such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\tau}{\Longrightarrow} \langle C', s'\rangle$. Moreover, if $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow\!\!\!\!\!|} P$ is a complete micro-computation, then $P \gtrsim [\![\langle C', s'\rangle]\!]$.*

Proof The proofs of the two statements are similar, so let us assume $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow\!\!\!\!\!|} P$, where $\alpha \in Act_{Env} \cup \{\tau\}$.

Now, if $\overset{\alpha}{\Longrightarrow\!\!\!\!\!|}$ is a simple relay move[8] or a transaction prefix, we have the result by Lemma D.3 or Lemma D.6, respectively. Otherwise, $\overset{\alpha}{\Longrightarrow\!\!\!\!\!|}$ starts with the locking step of some transaction and proceeds with an interleaving of transaction and relay moves. By repeated applications of Lemma D.5, we can bring all relay moves in front position to obtain, for some $n \geq 1$, a computation of the form:

$$[\![\langle C, s\rangle]\!] = P_0 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, P$$

where each $P_i \xrightarrow{\tau} P_{i+1}$ is a relay move and $P_n \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, P$ is a transaction prefix. Then, by Corollary D.4, we have a sequence of administrative moves for $\langle C, s\rangle$:

$$\langle C, s\rangle = \langle C_0, s\rangle \xrightarrow{\tau} \cdots \xrightarrow{\tau} \langle C_n, s\rangle$$

such that for each $i \geq 1$, $P_i \gtrsim [\![\langle C_i, s\rangle]\!]$. In particular $P_n \gtrsim [\![\langle C_n, s\rangle]\!]$. Then, since $P_n \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, P$ is a transaction prefix, there must exist Q such that $[\![\langle C_n, s\rangle]\!] \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, Q$ is a transaction prefix and $P \gtrsim Q$. By Lemma D.6, $[\![\langle C_n, s\rangle]\!] \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, Q$ implies either $Q \approx [\![\langle C_n, s\rangle]\!]$, in which case $\alpha = \tau$ and $\langle C, s\rangle \overset{\tau}{\Longrightarrow} \langle C_n, s\rangle$ is the required reflecting computation for $[\![\langle C, s\rangle]\!] \overset{\tau}{\Longrightarrow\!\!\!\!\!|}\, P$, since $P \gtrsim Q \approx [\![\langle C_n, s_n\rangle]\!]$, or there exist C', s' such that $\langle C_n, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$ and:

1. if $[\![\langle C_n, s\rangle]\!] \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, Q$ is an incomplete transaction, then $Q \overset{\hat{\tau}}{\Longrightarrow} [\![\langle C', s'\rangle]\!]$ and $Q \approx [\![\langle C', s'\rangle]\!]$;
2. if $[\![\langle C_n, s\rangle]\!] \overset{\alpha}{\Longrightarrow\!\!\!\!\!|}\, Q$ is a complete transaction, then $Q \equiv [\![\langle C', s'\rangle]\!]$.

In both cases $\langle C, s\rangle \overset{\tau}{\Longrightarrow} \langle C_n, s\rangle \xrightarrow{\alpha} \langle C', s'\rangle$ is the required reflecting computation. In case 1. we have $P \approx [\![\langle C', s'\rangle]\!]$ and in case 2. we have $P \gtrsim [\![\langle C', s'\rangle]\!]$.

□

Corollary D.8 (Sequences of micro-computations are reflected by the translation)

If $[\![\langle C, s\rangle]\!] \overset{\alpha_1}{\Longrightarrow\!\!\!\!\!|}\, P_1 \cdots \overset{\alpha_n}{\Longrightarrow\!\!\!\!\!|}\, P_n$ *is a sequence of micro-computations, then there exist* $C_1, \ldots, C_n$ *and* $s_1, \ldots s_n$ *such that* $\langle C, s\rangle \overset{\widetilde{\alpha_1}}{\Longrightarrow} \langle C_1, s_1\rangle \cdots \overset{\widetilde{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$, *where for each* i, $1 \leq i < n$, $P_i \gtrsim [\![\langle C_i, s_i\rangle]\!]$ *and:*

1. *If* $P_{n-1} \overset{\alpha_n}{\Longrightarrow\!\!\!\!\!|}\, P_n$ *is a complete micro-computation, then* $P_n \gtrsim [\![\langle C_n, s_n\rangle]\!]$;
2. *If* $P_{n-1} \overset{\alpha_n}{\Longrightarrow\!\!\!\!\!|}\, P_n$ *is not complete, then* $P_n \approx [\![\langle C_n, s_n\rangle]\!]$.

[8]Note that $\overset{\alpha}{\Longrightarrow\!\!\!\!\!|}$ cannot be a sequence of several relay moves, since that would not constitute a single micro-computation.

Proof By induction on n. If $n = 1$ we have the result by Proposition D.7. Suppose now that $n > 1$. By induction, $[\![\langle C, s\rangle]\!] = P_0 \stackrel{\alpha_1}{|\!\Longrightarrow} P_1 \cdots \stackrel{\alpha_{n-1}}{|\!\Longrightarrow} P_{n-1}$ implies that there exist $C_1, \ldots, C_{n-1}$ and $s_1, \ldots s_{n-1}$ such that $\langle C, s\rangle \stackrel{\widehat{\alpha_1}}{\Longrightarrow} \langle C_1, s_1\rangle \cdots \stackrel{\widehat{\alpha_{n-1}}}{\Longrightarrow} \langle C_{n-1}, s_{n-1}\rangle$, where for each i, $1 \leq i < n$, $P_i \gtrsim [\![\langle C_i, s_i\rangle]\!]$ (indeed, since $P_{n-2} \stackrel{\alpha_{n-1}}{|\!\Longrightarrow} P_{n-1}$ is complete, by 1. we have $P_{n-1} \gtrsim [\![\langle C_{n-1}, s_{n-1}\rangle]\!]$). Then, since $P_{n-1} \stackrel{\alpha_n}{|\!\Longrightarrow} P_n$, we deduce either $P_n \gtrsim [\![\langle C_{n-1}, s_{n-1}\rangle]\!]$ (this is possible if $\alpha_n = \tau$ and $P_{n-1} \stackrel{\tau}{|\!\Longrightarrow} P_n$ does not contain transaction moves and is simulated by inaction), or $[\![\langle C_{n-1}, s_{n-1}\rangle]\!] \stackrel{\alpha_n}{|\!\Longrightarrow} Q$, for some Q such that $P_n \gtrsim Q$. In the first case, we have the reflecting computation $\langle C, s\rangle \stackrel{\widehat{\alpha_1}}{\Longrightarrow} \langle C_1, s_1\rangle \cdots \stackrel{\widehat{\alpha_{n-1}}}{\Longrightarrow} \langle C_{n-1}, s_{n-1}\rangle \stackrel{\widehat{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$, where $\langle C_{n-1}, s_{n-1}\rangle \stackrel{\widehat{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$ is the empty computation and thus $\langle C_{n-1}, s_{n-1}\rangle = \langle C_n, s_n\rangle$. In the second case, by Proposition D.7 we have either $[\![\langle C_{n-1}, s_{n-1}\rangle]\!] \approx Q$, in which case the reflecting computation is again $\langle C, s\rangle \stackrel{\widehat{\alpha_1}}{\Longrightarrow} \langle C_1, s_1\rangle \cdots \stackrel{\widehat{\alpha_{n-1}}}{\Longrightarrow} \langle C_{n-1}, s_{n-1}\rangle \stackrel{\widehat{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$ where $\langle C_{n-1}, s_{n-1}\rangle \stackrel{\widehat{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$ is the empty computation, or there exist C_n, s_n such that $\langle C_{n-1}, s_{n-1}\rangle \stackrel{\alpha_n}{\Longrightarrow} \langle C_n, s_n\rangle$ and $Q \approx [\![\langle C_n, s_n\rangle]\!]$, in which case the reflecting computation is $\langle C, s\rangle \stackrel{\widehat{\alpha_1}}{\Longrightarrow} \langle C_1, s_1\rangle \cdots \stackrel{\widehat{\alpha_{n-1}}}{\Longrightarrow} \langle C_{n-1}, s_{n-1}\rangle \stackrel{\widehat{\alpha_n}}{\Longrightarrow} \langle C_n, s_n\rangle$, where $P_n \gtrsim Q \approx [\![\langle C_n, s_n\rangle]\!]$. □

Proposition D.9 (Unique decomposition into micro-computations)

Let $\langle C, s\rangle$ be a well-formed configuration and $\alpha \in Act_{Env} \cup \{\tau\}$. Then, each computation $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$ may be uniquely decomposed into a sequence of micro-computations of the form $[\![\langle C, s\rangle]\!] = P_0 \stackrel{\tau}{|\!\Longrightarrow} P_1 \cdots P_i \stackrel{\alpha}{|\!\Longrightarrow} P_{i+1} \cdots P_{n-1} \stackrel{\tau}{|\!\Longrightarrow} P_n = P$, where $0 \leq i < n$.

Proof By induction on the length k of the computation $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$. Note that the first step of $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$ is necessarily a τ-move. Then, if $k = 1$ the computation reduces to $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\longrightarrow} P$: this is either a relay move, which is by itself a complete micro-computation, or the locking step of a transaction, which is a proper transaction prefix, and hence a - non complete - micro-computation.

Suppose now $k > 1$. If the first step is a relay move, the computation has the form $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\longrightarrow} [\![\langle C_0, s_0\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$. By induction $[\![\langle C_0, s_0\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$ has a unique decomposition into a sequence of micro-computations $[\![\langle C_0, s_0\rangle]\!] \stackrel{\tau}{|\!\Longrightarrow} P_1 \cdots P_i \stackrel{\alpha}{|\!\Longrightarrow} P_{i+1} \cdots P_{n-1} \stackrel{\tau}{|\!\Longrightarrow} P_n = P$. Then $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\longrightarrow} [\![\langle C_0, s_0\rangle]\!] \stackrel{\tau}{|\!\Longrightarrow} P_1 \cdots P_i \stackrel{\alpha}{|\!\Longrightarrow} P_{i+1} \cdots P_{n-1} \stackrel{\tau}{|\!\Longrightarrow} P_n = P$ is the required decomposition. Suppose now the first step of $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$ is the locking step of a transaction. Then the computation has the form $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\longrightarrow} P_0 \stackrel{\alpha}{\Longrightarrow} P$. Consider the longest prefix $P_0 \stackrel{\tau}{\longrightarrow} P_1 \cdots P_j \stackrel{\alpha}{\longrightarrow} P_{j+1} \cdots P_{m-1} \stackrel{\tau}{\longrightarrow} P_m$ of the computation $P_0 \stackrel{\alpha}{\Longrightarrow} P$ such that $P_1, \ldots, P_m$ are locked (this prefix could also have the form $P_0 \stackrel{\tau}{\longrightarrow} P_1 \cdots P_{m-1} \stackrel{\tau}{\longrightarrow} P_m$, but we omit considering this case since it is treated similarly).

If $P_m = P$ then $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\longrightarrow} P_0 \stackrel{\tau}{\longrightarrow} P_1 \cdots P_j \stackrel{\alpha}{\longrightarrow} P_{j+1} \cdots P_{m-1} \stackrel{\tau}{\longrightarrow} P_m = P$ is the required (non complete) micro-computation. Otherwise, there exists an unlocked P_{m+1} such

that $P_m \xrightarrow{\tau} P_{m+1} \stackrel{\tau}{\Longrightarrow} P$. Then $[\![\langle C, s\rangle]\!] \xrightarrow{\tau} P_0 \cdots P_j \xrightarrow{\alpha} P_{j+1} \cdots P_m \xrightarrow{\tau} P_{m+1}$ constitutes a complete micro-computation $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{|\!\Longrightarrow} P_{m+1}$. Hence $P_{m+1} \equiv [\![\langle C', s'\rangle]\!]$ for some C', s'. Then, corresponding to the computation $P_{m+1} \stackrel{\tau}{\Longrightarrow} P$, $[\![\langle C', s'\rangle]\!]$ has a computation of the same length $[\![\langle C', s'\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q$, for some $Q \equiv P$. By induction $[\![\langle C', s'\rangle]\!] \stackrel{\tau}{\Longrightarrow} Q$ has a unique decomposition into a sequence of micro-computations $[\![\langle C', s'\rangle]\!] \stackrel{\tau}{|\!\Longrightarrow} Q_1 \cdots Q_{h-1} \stackrel{\tau}{|\!\Longrightarrow} Q_h = Q$. Let $P_{m+1} \stackrel{\tau}{|\!\Longrightarrow} R_1 \cdots R_{h-1} \stackrel{\tau}{|\!\Longrightarrow} R_h = P$ be the corresponding sequence of micro-computations of P_{m+1}. Then the required decomposition for the whole computation is:

$$[\![\langle C, s\rangle]\!] \stackrel{\alpha}{|\!\Longrightarrow} P_{m+1} \stackrel{\tau}{|\!\Longrightarrow} R_1 \cdots \stackrel{\tau}{|\!\Longrightarrow} R_h = P$$

□

Lemma 3.4 [Process transitions are reflected by the translation]

Let $\langle C, s\rangle$ be a well-formed configuration. Then:

1. If $[\![\langle C, s\rangle]\!] \stackrel{\alpha}{\Longrightarrow} P$, $\alpha \in Act_{Env}$, then $\exists\, C', s'$ such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\alpha}{\Longrightarrow} \langle C', s'\rangle$.
2. If $[\![\langle C, s\rangle]\!] \stackrel{\tau}{\Longrightarrow} P$, then either $P \approx [\![\langle C, s\rangle]\!]$ or $\exists\, C', s'$ such that $P \approx [\![\langle C', s'\rangle]\!]$ and $\langle C, s\rangle \stackrel{\tau}{\Longrightarrow} \langle C', s'\rangle$.

Proof Immediate consequence of Proposition D.9 and Corollary D.8. □

Adapting the proofs to the second translation

All the above proofs may be adapted, with some small modifications, to hold for the alternative translation of Figure 5. Let us briefly discuss the required changes.

Note first that, if one adopts the translation of Figure 5, an image derivative may perform, in addition to relay moves and transactions, also silent moves which transform a conditional ($\texttt{if}\ v\ \texttt{then}\ [\![C_1]\!]\ \texttt{else}\ [\![C_2]\!]$) into one of its branches depending on the value v. Such moves, which we call *steering moves*, occur in the simulation of conditionals and loops, after the semaphore has been released, and are reflected by the empty computation in the source program. If P is an image-derivative, let us call a move $P \xrightarrow{\tau} P'$ *autonomous* if it is either a relay move or a steering move. Then, by replacing relay moves by autonomous moves in the definition of the expansion preorder, and adapting correspondingly the proofs of Proposition D.7, Proposition D.9 and Lemma D.6 (whose statement also needs to be adapted by replacing $P = P'$ by $P \gtrsim P'$), one may extend the various results to the translation of Figure 5.

E Proofs of Proposition 3.11 and Theorem 3.13

Proposition 3.11 [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 $\langle C, s\rangle \backslash Env_H \sim_H \langle D, s\rangle$.

Proof *Proof of* $\Rightarrow$: Let $C \simeq_L D$. We want to show that for any state s we have both $\langle C, s\rangle \sim_H \langle D, s\rangle \backslash Env_H$ and $\langle C, s\rangle \backslash Env_H \sim_H \langle D, s\rangle$. Consider the following relation:

$$\begin{aligned}\mathcal{S} = \ & \{\, (\, \langle C_1, s_1\rangle, \langle C_2, s_2\rangle \backslash Env_H\,) \mid C_1 \simeq_L C_2\,,\ s_1 =_L s_2 \ \text{ and } \ \langle C_i, s_i\rangle \ \text{ is well-formed} \,\} \cup \\ & \{\, (\, \langle C_1, s_1\rangle \backslash Env_H, \langle C_2, s_2\rangle\,) \mid C_1 \simeq_L C_2\,,\ s_1 =_L s_2 \ \text{ and } \ \langle C_i, s_i\rangle \ \text{ is well-formed} \,\}\end{aligned}$$

Note that $\mathcal{S}$ is symmetric and for any state s such that $\langle C, s\rangle$ and $\langle D, s\rangle$ are well-formed, the two pairs $(\langle C, s\rangle, \langle D, s\rangle \backslash Env_H)$ and $(\langle C, s\rangle \backslash Env_H, \langle D, s\rangle)$ belong to $\mathcal{S}$. We show now that $\mathcal{S}$ is a bisimulation up to high. Since well-formedness is preserved by execution, we shall not mention it further in the proof. Take a pair $(\, \langle C_1, s_1\rangle, \langle C_2, s_2\rangle \backslash Env_H\,)$ in $\mathcal{S}$.

Let us start with the case where $\langle C_1, s_1\rangle$ moves first.

- Suppose that $\langle C_1, s_1\rangle \xrightarrow{\tau} \langle C_1', s_1'\rangle$. This means that $\langle C_1, s_1\rangle \rightarrow \langle C_1', s_1'\rangle$. Since by hypothesis $C_1 \simeq_L C_2$ and $s_1 =_L s_2$, this implies $\langle C_2, s_2\rangle \mapsto \langle C_2', s_2'\rangle$ for some C_2', s_2' such that $C_1' \simeq_L C_2'$ and $s_1' =_L s_2'$. This means that $\langle C_2, s_2\rangle \overset{\tau}{\longmapsto}_H \langle C_2', s_2'\rangle$. Hence $\langle C_2, s_2\rangle \backslash Env_H \overset{\tau}{\longmapsto}_H \langle C_2', s_2'\rangle \backslash Env_H$, where $(\langle C_1', s_1'\rangle, \langle C_2', s_2'\rangle \backslash Env_H)$ is again in $\mathcal{S}$.
- Suppose now $\langle C_1, s_1\rangle \xrightarrow{in_X v} \langle C_1', s_1'\rangle$. By Remark 3.2, $C_1' = C_1$ and $s_1' = s_1[v/X]$. Assume first that $X \in H$. Then $s_1[v/X] =_L s_1 =_L s_2$, so $\langle C_2, s_2\rangle \backslash Env_H$ can reply by the empty move. Suppose now $X \in L$. By Remark 3.2, $\langle C_2, s_2\rangle \xrightarrow{in_X v} \langle C_2, s_2[v/X]\,\rangle$. Since $in_X \notin Env_H$, $\langle C_2, s_2\rangle \backslash Env_H \xrightarrow{in_X v} \langle C_2, s_2[v/X]\rangle \backslash Env_H$, which is the required matching move since $s_1 =_L s_2$ implies $s_1[v/X] =_L s_2[v/X]$.
- The case where $\langle C_1, s_1\rangle \xrightarrow{\overline{out_X} v} \langle C_1', s_1'\rangle$ is similar (and simpler, since $s_1' = s_1$).

Consider now the case where $\langle C_2, s_2\rangle \backslash Env_H$ moves first.

- Suppose that $\langle C_2, s_2\rangle \backslash Env_H \xrightarrow{\alpha} \langle C_2', s_2'\rangle \backslash Env_H$. This transition is derived by rule (Res-Op) from $\langle C_2, s_2\rangle \xrightarrow{\alpha} \langle C_2', s_2'\rangle$, $subj(\alpha) \notin Env_H$. Note that either $\alpha = \tau$ or $subj(\alpha) \in Env_L$. In the first case we have $\langle C_2, s_2\rangle \xrightarrow{\tau} \langle C_2', s_2'\rangle$. Since $C_1 \simeq_L C_2$ and $s_1 =_L s_2$, this implies $\langle C_1, s_1\rangle \mapsto \langle C_1', s_1'\rangle$, and thus $\langle C_1, s_1\rangle \overset{\tau}{\longmapsto}_H \langle C_1', s_1'\rangle$ for some C_1', s_1' such that $C_1' \simeq_L C_2'$ and $s_1' =_L s_2'$. In the second case $subj(\alpha) \in Env_L$. Suppose for instance $\langle C_2, s_2\rangle \backslash Env_H \xrightarrow{in_X v} \langle C_2, s_2[v/X]\rangle \backslash Env_H$. By Remark 3.2, $\langle C_1, s_1\rangle \xrightarrow{in_X v} \langle C_1, s_1[v/X]\,\rangle$. From $s_1 =_L s_2$ we deduce $s_1[v/X] =_L s_2[v/X]$, so this is the required matching move. The case where $\alpha = \overline{out_X} v$ is similar.

This ends the "only if" part of the proof.

Proof of $\Leftarrow$: Suppose $\langle C, s\rangle \sim_H \langle D, s\rangle\backslash Env_H$ and $\langle C, s\rangle\backslash Env_H \sim_H \langle D, s\rangle$ for any state s. Define the relation $\mathcal{S}$ as:

$$\mathcal{S} = \{\, (C_1, C_2) \mid \forall \text{ state } t : \ (\langle C_1, t\rangle \sim_H \langle C_2, t\rangle\backslash Env_H \ \wedge\ \langle C_1, t\rangle\backslash Env_H \sim_H \langle C_2, t\rangle) \,\}$$

The relation $\mathcal{S}$ is symmetric and contains the pair (C, D). We show that $\mathcal{S}$ is a L-bisimulation. Take $(C_1, C_2) \in \mathcal{S}$. Let s_1, s_2 be such that $s_1 =_L s_2$. Since $\langle C_1, s_2\rangle \sim_H \langle C_2, s_2\rangle\backslash Env_H$, by repeatedly applying Corollary 3.9 we obtain $\langle C_1, s_1\rangle \sim_H \langle C_2, s_2\rangle\backslash Env_H$. Similarly, by Corollary 3.9, $\langle C_1, s_1\rangle\backslash Env_H \sim_H \langle C_2, s_1\rangle$ implies $\langle C_1, s_1\rangle\backslash Env_H \sim_H \langle C_2, s_2\rangle$.

- Suppose first $\langle C_1, s_1\rangle \xrightarrow{\tau} \langle C_1', s_1'\rangle$. Since $\langle C_1, s_1\rangle \sim_H \langle C_2, s_2\rangle\backslash Env_H$, there exist C_2', s_2' such that $\langle C_2, s_2\rangle\backslash Env_H \overset{\tau}{\longmapsto}_H \langle C_2', s_2'\rangle\backslash Env_H$ and $\langle C_1', s_1'\rangle \sim_H \langle C_2', s_2'\rangle\backslash Env_H$. We distinguish two cases, according to wether $\overset{\tau}{\longmapsto}_H$ is the empty move or a proper move.

  Case i). If $\overset{\tau}{\longmapsto}_H$ is the empty move, we have $\langle C_2', s_2'\rangle\backslash Env_H = \langle C_2, s_2\rangle\backslash Env_H$, hence by transitivity $\langle C_1', s_1'\rangle \sim_H \langle C_1, s_1\rangle$. Then $\langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_1, s_1\rangle\backslash Env_H$ by Property 3.6. Combining the various equalities we obtain: $\langle C_1', s_1'\rangle \sim_H \langle C_2', s_2'\rangle\backslash Env_H$ and $\langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_1, s_1\rangle\backslash Env_H \sim_H \langle C_2, s_2\rangle = \langle C_2', s_2'\rangle$.

  Case ii). If $\overset{\tau}{\longmapsto}_H$ is a proper τ-transition, it is deduced from $\langle C_2, s_2\rangle \xrightarrow{\tau} \langle C_2', s_2'\rangle$. Since $\langle C_1, s_1\rangle\backslash Env_H \sim_H \langle C_2, s_2\rangle$, this implies that there exist C_1'', s_1'' such that $\langle C_1, s_1\rangle\backslash Env_H \overset{\tau}{\longmapsto}_H \langle C_1'', s_1''\rangle\backslash Env_H$ and $\langle C_1'', s_1''\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle$. Now, by Lemma 3.10, $\langle C_1', s_1'\rangle \sim_H \langle C_2', s_2'\rangle\backslash Env_H$ implies $\langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle\backslash Env_H$, and $\langle C_1'', s_1''\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle$ implies $\langle C_1'', s_1''\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle\backslash Env_H$. Then by transitivity we obtain $\langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_1'', s_1''\rangle\backslash Env_H$.

  In both cases we have $\langle C_1', s_1'\rangle \sim_H \langle C_2', s_2'\rangle\backslash Env_H$ and $\langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle$. To obtain $(C_1', C_2') \in \mathcal{S}$, what is left to do is to replace s_1' and s_2' by an arbitrary state t in these equalities. Note that by Property 3.5 $s_1' =_L s_2'$, hence $\mathtt{dom}(s_1') = \mathtt{dom}(s_2')$.

  For any state t, let the low and high parts of t be denoted by t_L and t_H respectively, so that $t = t_L \cup t_H$. Using Lemmas 3.7 and 3.8 we may now update the low and high parts of s_1', s_2' so as to transform both of them into t. More precisely, starting from:

  $$\langle C_1', s_1'\rangle \sim_H \langle C_2', s_2'\rangle\backslash Env_H \quad\wedge\quad \langle C_1', s_1'\rangle\backslash Env_H \sim_H \langle C_2', s_2'\rangle$$

  we may update the low part of each s_i' using Lemma 3.7, so as to obtain:

  $$\langle C_1', t_L \cup s_{1H}'\rangle \sim_H \langle C_2', t_L \cup s_{2H}'\rangle\backslash Env_H \quad\wedge\quad \langle C_1', t_L \cup s_{1H}'\rangle\backslash Env_H \sim_H \langle C_2', t_L \cup s_{2H}'\rangle$$

  Using Lemma 3.8, we may now update the high part of each state to get:

  $$\langle C_1', t_L \cup t_H\rangle \sim_H \langle C_2', t_L \cup t_H\rangle\backslash Env_H \quad\wedge\quad \langle C_1', t_L \cup t_H\rangle\backslash Env_H \sim_H \langle C_2', t_L \cup t_H\rangle$$

  which is the required result.

- The case where $\langle C_2, s_2\rangle$ moves first is symmetric.

□

Theorem 3.13 [Bisimilarity up to high is preserved by the translation]

Let $R, R' \subseteq Env_H$. Then $\langle C, s\rangle\backslash R \sim_H \langle D, t\rangle\backslash R'$ implies $[\![\langle C, s\rangle\backslash R]\!] \approx_{\mathcal{H}} [\![\langle D, t\rangle\backslash R']\!]$, where $\mathcal{H} \stackrel{\text{def}}{=} \{\, get_X, put_X, in_X, out_X \mid X \in H \,\} \cup \{\, \texttt{lock}, \texttt{unlock}, \texttt{res}, \texttt{done} \,\}$.

Proof We show that the relation $\mathcal{S}$ defined as follows:

$$\mathcal{S} = \{\, (W_1, W_2) \mid \; W_i \approx [\![cfg_i]\!] \text{ for some } cfg_i = \langle C_i, s_i\rangle\backslash R_i \text{ such that } R_i \subseteq Env_H \text{ and } cfg_1 \sim_H cfg_2 \}$$

is a weak bisimulation up to $\mathcal{H}$. Clearly the pair $([\![\langle C, s\rangle\backslash R]\!], [\![\langle D, t\rangle\backslash R']\!])$ belongs to $\mathcal{S}$.

- Suppose $W_1 \xrightarrow{\tau} W_1'$. Since $W_1 \approx [\![cfg_1]\!]$, there exists P_1 such that $[\![cfg_1]\!] \stackrel{\hat{\tau}}{\Longrightarrow} P_1 \approx W_1'$. Now, if $P_1 = [\![cfg_1]\!]$, then $W_1' \approx W_1$ and W_2 may reply by staying idle. Otherwise, $[\![cfg_1]\!] \stackrel{\tau}{\Longrightarrow} P_1 \approx W_1'$. Then, by Lemma 3.4, we know that either $P_1 \approx [\![cfg_1]\!]$, in which case $W_1' \approx W_1$ and W_2 may reply by staying idle, or there exists cfg_1' such that $W_1' \approx [\![cfg_1']\!]$ and $cfg_1 \stackrel{\hat{\tau}}{\Longrightarrow} cfg_1'$. In this case, since $cfg_1 \sim_H cfg_2$, there exists cfg_2' such that $cfg_2 \stackrel{\hat{\tau}}{\Longrightarrow} cfg_2'$ and $cfg_1' \sim_H cfg_2'$. Then, by Lemma 3.3, we know that there exists P_2 such that $[\![cfg_2]\!] \stackrel{\hat{\tau}}{\Longrightarrow} P_2 \approx [\![cfg_2']\!]$. Since $W_2 \approx [\![cfg_2]\!]$, this implies that there exists W_2' such that $W_2 \stackrel{\hat{\tau}}{\Longrightarrow} W_2'$ and $W_2' \approx P_2 \approx [\![cfg_2']\!]$. Hence $(W_1', W_2') \in \mathcal{S}$ and we may conclude.
- Suppose now that $W_1 \xrightarrow{\alpha} W_1'$, where $\alpha \neq \tau$. Since $W_1 \approx [\![cfg_1]\!]$, there exists P_1 such that $[\![cfg_1]\!] \stackrel{\alpha}{\Longrightarrow} P_1 \approx W_1'$. Then, by Lemma 3.4, there exists cfg_1' such that $W_1' \approx [\![cfg_1']\!]$ and $cfg_1 \stackrel{\alpha}{\Longrightarrow} cfg_1'$. We can decompose $cfg_1 \stackrel{\alpha}{\Longrightarrow} cfg_1'$ as $cfg_1 \stackrel{\hat{\tau}}{\Longrightarrow} cfg_1'' \xrightarrow{\alpha} cfg_1''' \stackrel{\hat{\tau}}{\Longrightarrow} cfg_1'$. Since $cfg_1 \sim_H cfg_2$, cfg_2 may simulate this transition sequence either by a sequence $cfg_2 \stackrel{\hat{\tau}}{\Longrightarrow} cfg_2'' \xrightarrow{\alpha} cfg_2''' \stackrel{\hat{\tau}}{\Longrightarrow} cfg_2'$, where $cfg_1'' \sim_H cfg_2''$, $cfg_1''' \sim_H cfg_2'''$ and $cfg_1' \sim_H cfg_2'$, or possibly, in case $\alpha \in Env_H$, by a sequence of the form $cfg_2 \stackrel{\hat{\tau}}{\Longrightarrow} cfg_2'' \stackrel{\hat{\tau}}{\Longrightarrow} cfg_2'$, where $cfg_1'' \sim_H cfg_2''$ and $cfg_1' \sim_H cfg_2'$. By (repeated applications of) Lemma 3.3, we obtain in the first case $[\![cfg_2]\!] \stackrel{\alpha}{\Longrightarrow} P_2 \approx [\![cfg_2']\!]$, and in the second case $[\![cfg_2]\!] \stackrel{\hat{\tau}}{\Longrightarrow} P_2 \approx [\![cfg_2']\!]$. Since $W_2 \approx [\![cfg_2]\!]$, this implies that there exists W_2' such that $W_2' \approx P_2 \approx [\![cfg_2']\!]$ and either $W_2 \stackrel{\alpha}{\Longrightarrow} W_2'$ (in the first case) or $W_2 \stackrel{\hat{\tau}}{\Longrightarrow} W_2'$ (in the second case). Therefore $(W_1', W_2') \in \mathcal{S}$ and we may conclude.

□

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399