Noninterference for Concurrent Programs and Thread Systems *

Gérard Boudol and Ilaria Castellani

INRIA
2004 Route des Lucioles, BP 93
06902 Sophia-Antipolis – France

Abstract

We propose a type system to ensure the property of *noninterference* in a system of concurrent programs, described in a standard imperative language enriched with parallelism. Our proposal is in the line of some recent work by Irvine, Volpano and Smith. Our type system seems more natural and less restrictive than that originally presented by these authors for the concurrent case. Moreover, we show how to extend the language in order to formalise scheduling policies for systems of sequential threads. The type system is extended to the new constructs, and we show that noninterference still holds, while remaining in a nonprobabilistic setting.

1 Introduction

The aim of this paper is to study the property of secure information flow, and more specifically of *noninterference* (a notion first introduced by Goguen and Meseguer in [6]) in the setting of concurrency. Here noninterference means that variables of a given security level do not interfere with those of lower or incomparable levels. More precisely, the values of variables at some level are not dependent on the values of variables of higher or incomparable level. Noninterference is meant to model the absence of information flow from any security level σ_0 to another one σ_1, except when this is explicitly allowed, that is $\sigma_0 \leq \sigma_1$. Such information flow is considered *insecure*, as it amounts to the disclosure of secret information into the public domain. In this introduction, and in the examples given in the paper, the security levels will simply be high (H), or *secret*, and low (L), or *public*. However all results will be given for an arbitrary lattice of security levels. For some

*Research partially funded by the EU Working Group CONFER II and by the french RNRT Project MARVEL.

concrete examples using a more sophisticated lattice of security levels, see for instance [3].

Our starting point is the paper [21] by Volpano, Smith and Irvine, and the subsequent paper [18] by Smith and Volpano, where noninterference is enforced by means of a simple type system in an imperative language with security levels. The language considered in [21] is purely sequential, and is extended in [18] with asynchronous parallelism (interleaving). In such a language, insecure flow can be *explicit*, when assigning the value of a high variable to a low variable, or *implicit*, when testing the value of a high variable and then assigning to a low variable a value depending on the result of the test, for instance. In the approach of [21, 18], explicit flow is prevented by requiring that the level of the assigned variable be at least as high as that of the source variable, while implicit flow is prevented by asking that the level of the commands in the branches of a conditional (the level of a command being that of its lowest assigned variables) be at least as high as that of the tested variable. Implicit flow can also arise in while-loops, and is prevented by a similar condition on the type of the body of the loop.

In fact, because of while-loops, the definition of noninterference is more precise than what is stated above: it says that no change in the values of low-level variables should be observed as a consequence of a change in high-level variables, *provided that the program terminates successfully.* Using subscripts to explicitly indicate the security level of a variable, let us consider the following program, that terminates if $x_H \neq 0$ and loops forever (doing nothing) otherwise:

$$\texttt{while}\ x_H = 0\ \texttt{do nil}\ ;\ y_L := 1 \tag{1}$$

Should this program be accepted, that is, should it be typable? According to the above definition of noninterference the answer is "yes", since whenever the program terminates it produces the same value $y_L = 1$ for its low-level variable. Indeed, this program is typable in the original system of Volpano, Smith and Irvine [21], since the loop is typable and the sequential composition of typable programs is always typable.

However, accepting such a program leads to problems when parallelism is introduced in the language. These problems can be concisely described as "disguising information flow as control flow". Let us illustrate the problem by means of an example, which is a simplified version of the *PIN* example given by Smith and Volpano in [18]. In this example, given in Figure 1, three threads α, β and γ are run (asynchronously) in parallel. There are four variables, a high-level variable *PIN* tested by thread γ, two high-level variables t_α and t_β serving as "triggers" for threads α and β, and a low-level variable r written by α and β. As can be easily seen, with initial values $t_\alpha = t_\beta = \mathit{ff}$ the effect of this program is to copy the value of the secret variable *PIN* into the public variable r. The illicit information flow from

$$\gamma : \texttt{if } PIN = 0 \texttt{ then } t_\beta := tt \texttt{ else } t_\alpha := tt$$
$$\alpha : \texttt{while } t_\alpha \neq tt \texttt{ do nil} \ ; \ r := 0 \ ; \ t_\beta := tt$$
$$\beta : \texttt{while } t_\beta \neq tt \texttt{ do nil} \ ; \ r := 1 \ ; \ t_\alpha := tt$$

PIN, t_α, t_β : boolean variables of type H

r : boolean variable of type L

Figure 1: Information Flow through Control Flow

PIN to r is implemented through the *control flow* from γ to α or β. However, if we assume that a system of concurrent threads is typable provided each component is typable, as would be natural, this particular system is to be accepted.

To circumvent this problem, Smith and Volpano propose in [18] to forbid the use of high-level variables in guards of while-loops, and more specifically([1]) to accept only while-loops of low level. While ruling out the program in (1), and also the threads α and β of the PIN example, this solution seems a bit drastic. It excludes inoffensive programs such as `while` $x_H = 0$ `do nil`. We shall propose here a different solution to the problem raised by while-loops in the presence of parallelism, which allows this program to be typed, while ruling out the programs of example (1) and Figure 1. Our solution is based on the observation that a program such as

$$\texttt{while } x_H = 0 \texttt{ do nil}$$

should indeed be considered with some care in a concurrent setting, but only as a "guard", that is, as regards what may follow it. In the context of concurrent threads, if the control comes back to this `while` loop, this may be with a value for x_H different from 0, contrarily to what happens in a sequential setting. In other words, this program may observe the behaviour of other, concurrent components, in the course of their execution, and influence accordingly the behaviour of the thread in which it participates. Our aim is then to ensure a stronger form of noninterference, where the *course of values* – not just the final value – of a low-level variable does not depend upon the value of high variables. Typically, the program (1) is no longer interference-free in this stronger sense. In order to reject it, we introduce a refinement of the notion of type, and a new type system, where the level of

[1]assuming, in the case of an arbitrary lattice of security levels, that there is a lowest level $\perp$.

loop guards – the expressions tested by `while` loops – is taken into account in sequential composition. The basic idea is that assigning to a low variable must not depend upon consulting the value of a high variable. Then also conditional guards – the expressions tested by conditional statements – will be taken into account (we will see an example in the next section).

We will also examine the situation where a scheduling policy is in force in a thread system of sequential programs: we will introduce a couple of new programming primitives to describe formally such a situation, and show how to extend the type system to this new setting, where new interference phenomena arise. We show that our type system still ensures the noninterference property in this case, while imposing a less severe restriction than that prefigured in [18].

The rest of the paper is organised as follows. In Section 2 we introduce the language, its operational semantics and its type system. Section 3 presents the properties of typed programs, including subject reduction and noninterference. Finally, in Section 4 we consider the extended language with scheduling policies for sequential threads.

2 The Language and Type System

The language we consider is essentially that of [18] (where e stands for a boolean or arithmetic expression, whose syntax we do not detail here). We use the following two-level syntax, where U, V denote sequential programs, while P, Q denote general (concurrent) programs:

$$\begin{array}{rcl} U,\ V\ldots & ::= & \mathtt{nil} \ \mid\ x := e \ \mid\ U;V \ \mid\ \mathtt{if}\ e\ \mathtt{then}\ U\ \mathtt{else}\ V \\ & \mid & \mathtt{while}\ e\ \mathtt{do}\ U \\ P,\ Q\ldots & ::= & U \ \mid\ U;P \ \mid\ \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q \ \mid\ (P \parallel Q) \end{array}$$

Note that on the left of a sequential composition, we must have a sequential program. Thus programs of the form $(P \parallel Q); R$ are not allowed. With this restriction, our language is still more general than that of [18], which describes concurrent systems as collections of threads, thus allowing only top-level parallelism, while we allow the dynamic spawning of new threads (although in a limited way).

The operational semantics of the language is given in terms of transitions between configurations $(P, \mu) \to (P', \mu')$ where P, P' are programs and μ, μ' stand for *memories*, that is mappings from variables to values. These mappings are extended in the obvious way to expressions, whose evaluation is assumed to be atomic as in [18]. We use the notation $\mu[v/x]$ for memory update. The rules specifying the operational semantics of programs are presented in Figure 2. The semantics used here is a *small step semantics*, as

$$(\textsc{Assign-Op}) \quad \frac{}{(x := e, \mu) \to (\texttt{nil}, \mu[\mu(e)/x])}$$

$$(\textsc{Seq-Op1}) \quad \frac{(U, \mu) \to (U', \mu')}{(U; P, \mu) \to (U'; P, \mu')}$$

$$(\textsc{Seq-Op2}) \quad \frac{(P, \mu) \to (P', \mu')}{(\texttt{nil}; P, \mu) \to (P', \mu')}$$

$$(\textsc{Cond-Op1}) \quad \frac{\mu(e) = tt}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q, \mu) \to (P, \mu)}$$

$$(\textsc{Cond-Op2}) \quad \frac{\mu(e) \neq tt}{(\texttt{if } e \texttt{ then } P \texttt{ else } Q, \mu) \to (Q, \mu)}$$

$$(\textsc{While-Op1}) \quad \frac{\mu(e) = tt}{(\texttt{while } e \texttt{ do } U, \mu) \to (U; \texttt{while } e \texttt{ do } U, \mu)}$$

$$(\textsc{While-Op2}) \quad \frac{\mu(e) \neq tt}{(\texttt{while } e \texttt{ do } U, \mu) \to (\texttt{nil}, \mu)}$$

$$(\textsc{Par-Op1}) \quad \frac{(P, \mu) \to (P', \mu')}{(P \parallel Q, \mu) \to (P' \parallel Q, \mu')}$$

$$(\textsc{Par-Op2}) \quad \frac{(Q, \mu) \to (Q', \mu')}{(P \parallel Q, \mu) \to (P \parallel Q', \mu')}$$

Figure 2: Operational Semantics for Parallel Programs

---

opposed to the *big step semantics* of [21]([2]). The rules are fairly standard, and we shall not comment on them.

In the introduction we argued that, in a concurrent setting, the program (1) should be treated as another case of implicit information flow. Intuitively, when exiting a loop one gets some information about its guard; it seems then appropriate to require that what follows the loop – its "continuation" – have level at least as high as that of the loop guard. This will be the basic idea of our new type system, which is closely inspired by that given by Volpano et al. in [21] – however as suggested by the above Example (1) it will be more restrictive than that of [21] on the sequential sublanguage,

[2] In [18], the semantics is a mixture of small and big step semantics: transitions are given between configurations but there are two kinds of configurations, intermediate and final ones, suggesting that termination should be observed.

$$\text{(Nil)} \quad \frac{}{\Gamma \vdash \texttt{nil} : (\tau, \sigma)\; cmd}$$

$$\text{(Assign)} \quad \frac{\Gamma \vdash e : \tau, \quad \Gamma(x) = \tau\; var}{\Gamma \vdash x := e : (\tau, \sigma)\; cmd}$$

$$\text{(Seq)} \quad \frac{\Gamma \vdash U : (\tau, \sigma)\; cmd, \quad \Gamma \vdash P : (\tau', \sigma')\; cmd, \quad \sigma \leq \tau'}{\Gamma \vdash U\, ;\, P : (\tau \sqcap \tau', \sigma \sqcup \sigma')\; cmd}$$

$$\text{(Cond)} \quad \frac{\Gamma \vdash e : \theta, \quad \Gamma \vdash P_i : (\tau, \sigma)\; cmd, \quad \theta \leq \tau}{\Gamma \vdash \texttt{if}\; e\; \texttt{then}\; P_0\; \texttt{else}\; P_1 : (\tau, \theta \sqcup \sigma)\; cmd}$$

$$\text{(While)} \quad \frac{\Gamma \vdash e : \theta, \quad \Gamma \vdash U : (\tau, \sigma)\; cmd, \quad \theta \sqcup \sigma \leq \tau}{\Gamma \vdash \texttt{while}\; e\; \texttt{do}\; U : (\tau, \theta \sqcup \sigma)\; cmd}$$

$$\text{(Par)} \quad \frac{\Gamma \vdash P_i : (\tau, \sigma)\; cmd}{\Gamma \vdash P_0 \parallel P_1 : (\tau, \sigma)\; cmd}$$

$$\text{(Subtyping)} \quad \frac{\Gamma \vdash P : (\tau, \sigma)\; cmd, \quad \tau' \leq \tau, \quad \sigma \leq \sigma'}{\Gamma \vdash P : (\tau', \sigma')\; cmd}$$

Figure 3: Typing Rules for Concurrent Programs

because of our more detailed observation of programs, where an attacker is allowed to read the low part of the memory at any time.

The types of data and expressions are *security levels*, that is elements of a lattice $(\mathcal{T}, \leq)$. We denote the operations of meet and join respectively by $\sqcap$ and $\sqcup$. These types are ranged over by τ, $\sigma \ldots$ In the examples, the lattice of security levels will simply be $\{L, H\}$, with $L < H$. The types of variables (when used in the left-hand side of an assignment) are of the form $\tau\; var$. Our first point of departure from [21] concerns the types for programs. Type judgements in [21] are of the form $\Gamma \vdash P : \tau\; cmd$, where Γ is a mapping from variables to types of variables, i.e. elements of $\{\tau\; var \mid \tau \in \mathcal{T}\}$. The meaning of $\Gamma \vdash P : \tau\; cmd$ is that in the type environment Γ, the type τ is a *lower bound* for the level of the *assigned variables* of P. In line with this intuition, subtyping for programs is contravariant, that is $\tau\; cmd \leq \tau'\; cmd$ if $\tau' \leq \tau$. Thus for instance any program of type $H\; cmd$ can be downgraded to type $L\, cmd$. A program of type $H\; cmd$ is guaranteed not to contain any assignment to a low variable.

As we said previously, we have to take into account the level of loop guards in typing. To deal with possible divergence (and, as we shall see, with the scheduling of threads) we also have to record the level of the tested expression in conditional branching, which we regard as a guard as well. Thus we shall use here more refined types $(\tau, \sigma)\, cmd$, where the first component τ plays the same rôle as in the type $\tau\, cmd$, while the second component σ is the *guard type*, an *upper bound* on the level of the loop and conditional guards occurring in a program. Accordingly, the subtyping for programs is contravariant in its first component and covariant in the second:

$$(\tau, \sigma)\, cmd \leq (\tau', \sigma')\, cmd \quad \text{if} \quad \tau' \leq \tau \quad \text{and} \quad \sigma \leq \sigma'$$

The guard type will be set up by while-loops and conditional branchings, and looked up by sequential composition. The complete type system for programs is shown in Figure 3, where we assume given a system allowing us to infer judgements $\Gamma \vdash e : \tau$ about the security level of an expression. These are used in the (Assign), (Cond) and (While) rules. Notice that the guard type plays no particular rôle in rules (Nil) and (Assign), which are plain adaptations of the ones in [21]. As in that paper, the side condition $\theta \leq \tau$ in the rules (Cond) and (while) is meant to avoid implicit insecure flow. Let us comment a little on the rules for conditional branching, while-loops and sequential composition, which are the main novelty w.r.t. [21, 18]. As explained, the guard type is at least θ for a conditional branching or a while-loop testing an expression of level θ, and from then onwards it should remain at least θ to prevent concatenation with programs of a lower or incomparable level. Rule (Seq) is precisely designed to avoid sequencing "low" assignments after a program with "high" guards. This rules out the kind of implicit flow exhibited by the program (1). It also rules out the (semantically equivalent, in a sequential setting) program

$$\begin{array}{lll} \texttt{if } x_H = 0 & \texttt{then} & \texttt{while } tt \texttt{ do nil} \\ & \texttt{else} & \texttt{nil}\,; \\ y_L := 1 & & \end{array} \tag{2}$$

which is not interference-free in our sense. Indeed, this example shows why the level of conditional guards should be recorded as a guard type. The side condition $\sigma \leq \tau$ in the rule (While) is needed for the preservation of types during execution, as one can see from the unfolding of `while` e `do` P into `if` e `then` P;`while` e `do` P `else nil`. Notice that if P_0 and P_1 are both typable in the typing context Γ, say with respective types $(\tau_0, \sigma_0)\, cmd$ and $(\tau_1, \sigma_1)\, cmd$, then their parallel composition $(P_0 \parallel P_1)$ is typable, since both P_0 and P_1 have type $(\tau_0 \sqcap \tau_1, \sigma_0 \sqcup \sigma_1)\, cmd$, thanks to the (Subtyping) rule.

3 Properties of Typed Programs

In this section we prove some desired properties of our type system. The first property, *subject reduction*, states that types are preserved along execution.

Theorem 3.1 (Subject Reduction)
If $\Gamma \vdash P : (\tau, \sigma)\ cmd$ *and* $(P, \mu) \rightarrow (P', \mu')$*, then* $\Gamma \vdash P' : (\tau, \sigma)\ cmd$*.*

Proof: By induction on the inference of $\Gamma \vdash P : (\tau, \sigma)\ cmd$, and then case analysis on the last rule used in this inference. If this rule is the subtyping rule, we simply use the induction hypothesis. We examine the main cases:

(Assign). Here we have $P = x := e$ with $\Gamma \vdash e : \tau$ and $\Gamma(x) = \tau\ var$. Moreover, it must be that $P' = \mathtt{nil}$ and $\mu' = \mu[\mu(e)/x]$. The theorem is trivial in this case, since $\Gamma \vdash \mathtt{nil} : (\tau, \sigma)\ cmd$.

(Seq). Here $P = U; Q$ with $\Gamma \vdash U : (\tau_1, \sigma_1)\ cmd$, $\Gamma \vdash Q : (\tau_2, \sigma_2)\ cmd$, $\tau = \tau_1 \sqcap \tau_2$ and $\sigma = \sigma_1 \sqcup \sigma_2$ with $\sigma_1 \leq \tau_2$. There are two possibilities for a transition $(P, \mu) \rightarrow (P', \mu')$. If this results from $(U, \mu) \rightarrow (U', \mu')$, by means of (Seq-Op1), we have $P' = U'; Q$, where $\Gamma \vdash U' : (\tau_1, \sigma_1)\ cmd$ by induction, and we conclude using the rule (Seq). If (Seq-Op2) is used, we have $U = \mathtt{nil}$ with $(Q, \mu) \rightarrow (P', \mu')$. By induction $\Gamma \vdash P' : (\tau_2, \sigma_2)\ cmd$, and we use the subtyping rule to conclude.

(While). Here $P = \mathtt{while}\ e\ \mathtt{do}\ U$ and $\Gamma \vdash U : (\tau, \sigma')\ cmd$, with $\sigma = \theta \sqcup \sigma'$ and $\sigma \leq \tau$. If $\mu(e) = tt$ we have $P' = U; P$, and since $\sigma' \leq \tau$ we may use the rule (Seq) to conclude $\Gamma \vdash P' : (\tau, \sigma)\ cmd$. Otherwise $P' = \mathtt{nil}$, and this case is trivial.

All the other cases are similar (in the case of (Cond) we use subtyping). □

We shall use the following assumptions about expressions:

Assumption 3.2 (Termination of Expression Evaluation)
For any memory μ *and expression* e*, the value* $\mu(e)$ *is defined.*

Notice that this implies for instance that for any sequential program $U \neq \mathtt{nil}\,; \cdots ; \mathtt{nil}$ and for any μ there exist U' and μ' such that $(U, \mu) \rightarrow (U', \mu')$.

Assumption 3.3 (Simple Security)
If $\Gamma \vdash e : \tau$*, then every variable occurring in* e *has type* $\tau'\ var$ *in* Γ*, with* $\tau' \leq \tau$*.*

We introduce now a notion of equality on memories, depending on a given type environment Γ and a given set $\mathcal{L} \subseteq \mathcal{T}$ of security levels, which are to be understood as "low" security levels. In the following, $\mathcal{L}$ will always denote a *downward-closed* set of security levels, that is a set satisfying

$$\tau \in \mathcal{L}\ \&\ \sigma \leq \tau \ \Rightarrow\ \sigma \in \mathcal{L}$$

Two memories are $(\Gamma,\mathcal{L})$-equal if they coincide on variables to which Γ assigns a level in $\mathcal{L}$. Intuitively, such memories are indistinguishable for an observer which has access only to information of level in $\mathcal{L}$.

Definition 3.4 **($(\Gamma,\mathcal{L})$-Equality of Memories)**
$\mu =_\Gamma^\mathcal{L} \nu \;\Leftrightarrow_{\text{def}}\; \forall x.\, \Gamma(x) = \tau\, var \;\&\; \tau \in \mathcal{L} \;\Rightarrow\; \mu(x) = \nu(x)$.

In the examples we implicitly used the particular case $=_\Gamma^{\{L\}}$, which is denoted $\sim_\Gamma$ in [18]. As it has become standard, we use the notion of *bisimulation* [11, 13] to (state and) establish the security properties of concurrent programs. We shall actually use several kinds of bisimulations. To define these notions, it is convenient to introduce some notations. First, we denote by $\twoheadrightarrow$ the reflexive closure of $\rightarrow$, that is:

$$(P,\mu) \twoheadrightarrow (P',\mu') \;\Leftrightarrow_{\text{def}}\; P' = P \;\&\; \mu' = \mu \quad \text{or} \quad (P,\mu) \rightarrow (P',\mu')$$

As usual, $\xrightarrow{*}$ is the reflexive and transitive closure of $\rightarrow$. We say that P' is a *derivative* of P, denoted $P \rightsquigarrow P'$, if for some μ and μ' we have $(P,\mu) \xrightarrow{*} (P',\mu')$. Then, given a typing context Γ and a set $\mathcal{L}$ of (low) security levels, we denote by $\mathcal{P}^{\Gamma,\mathcal{L}}$ the set of "(semantically) high" programs, that is programs that never modify the low part of the memory:

$$P \in \mathcal{P}^{\Gamma,\mathcal{L}} \;\Leftrightarrow_{\text{def}}\; \forall P',\mu,P'',\mu'.\; P \rightsquigarrow P' \;\&\; (P',\mu) \xrightarrow{*} (P'',\mu') \;\Rightarrow\; \mu' =_\Gamma^\mathcal{L} \mu$$

Clearly, if $P \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and P' is a derivative of P then $P' \in \mathcal{P}^{\Gamma,\mathcal{L}}$.

Definition 3.5 **($(\Gamma,\mathcal{L})$-Bisimulation)**
A relation $\mathcal{R}$ on configurations is a $(\Gamma,\mathcal{L})$-bisimulation if $\mathcal{R}$ is symmetric and $(P,\mu)\,\mathcal{R}\,(Q,\nu)$ implies

(i) $\mu =_\Gamma^\mathcal{L} \nu$

(ii) $(P,\mu) \rightarrow (P',\mu') \;\Rightarrow\; \exists Q',\nu'.\; (Q,\nu) \twoheadrightarrow (Q',\nu') \;\&\; (P',\mu')\,\mathcal{R}\,(Q',\nu')$

The relation $\mathcal{R}$ is a *strong* $(\Gamma,\mathcal{L})$-bisimulation if it satisfies the conditions above, where $\twoheadrightarrow$ is replaced by $\rightarrow$. The $(\Gamma,\mathcal{L})$*-bisimulation equivalence* on configurations, noted $\approx_\Gamma^\mathcal{L}$, (resp. the *strong* $(\Gamma,\mathcal{L})$*-bisimulation equivalence*, noted $\sim_\Gamma^\mathcal{L}$) is the largest $(\Gamma,\mathcal{L})$-bisimulation (resp. the largest strong $(\Gamma,\mathcal{L})$-bisimulation).

Clearly, we have $\sim_\Gamma^\mathcal{L} \subseteq \approx_\Gamma^\mathcal{L}$, and the inclusion is strict, since for instance, for any memories μ and ν such that $\mu =_\Gamma^\mathcal{L} \nu$, we have:

$$(\texttt{nil},\mu) \approx_\Gamma^\mathcal{L} (\texttt{while}\ tt\ \texttt{do}\ \texttt{nil},\nu)$$

This is a consequence of the following:

Lemma 3.6 (Bisimilarity of High Programs)
The relation

$$\mathcal{R}_0^{\Gamma,\mathcal{L}} = \{((P,\mu),(Q,\nu)) \mid P \in \mathcal{P}^{\Gamma,\mathcal{L}}, Q \in \mathcal{P}^{\Gamma,\mathcal{L}} \ \&\ \mu =_\Gamma^\mathcal{L} \nu\}$$

is a $(\Gamma,\mathcal{L})$*-bisimulation.*

Proof: This is immediate, because if $(P,\mu) \to (P',\mu')$ then $P' \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and $\mu' =_\Gamma^\mathcal{L} \mu =_\Gamma^\mathcal{L} \nu$. □

For technical reasons we introduce one last notion of bisimulation:

Definition 3.7 (Quasi-Strong $(\Gamma,\mathcal{L})$-Bisimulation)
A relation $\mathcal{R}$ on configurations is a quasi-strong $(\Gamma,\mathcal{L})$-bisimulation if $\mathcal{R}$ is symmetric and $(P,\mu)\,\mathcal{R}\,(Q,\nu)$ implies

- **(i)** $\mu =_\Gamma^\mathcal{L} \nu$
- **(ii)** $P \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and $Q \in \mathcal{P}^{\Gamma,\mathcal{L}}$, or
  $(P,\mu) \to (P',\mu') \ \Rightarrow\ \exists Q',\nu'.\ (Q,\nu) \to (Q',\nu')\ \&\ (P',\mu')\,\mathcal{R}\,(Q',\nu')$

The *quasi-strong* $(\Gamma,\mathcal{L})$*-bisimulation equivalence* on configurations, noted $\simeq_\Gamma^\mathcal{L}$, is the largest quasi-strong $(\Gamma,\mathcal{L})$-bisimulation.

It should be clear that a quasi-strong $(\Gamma,\mathcal{L})$-bisimulation is also a $(\Gamma,\mathcal{L})$-bisimulation, and that a strong $(\Gamma,\mathcal{L})$-bisimulation is also a quasi-strong $(\Gamma,\mathcal{L})$-bisimulation, and therefore we have $\sim_\Gamma^\mathcal{L} \subseteq \simeq_\Gamma^\mathcal{L} \subseteq \approx_\Gamma^\mathcal{L}$. As in [16], we shall say, by abuse of language, that a relation $\mathcal{S}$ *on programs* is a (strong, quasi-strong) $(\Gamma,\mathcal{L})$-bisimulation if the relation

$$\{((P,\mu),(Q,\nu)) \mid P\,\mathcal{S}\,Q \ \&\ \mu =_\Gamma^\mathcal{L} \nu\}$$

is a (strong, quasi-strong) $(\Gamma,\mathcal{L})$-bisimulation. For instance, Lemma 3.6 shows that the relation $\mathcal{S}_0^{\Gamma,\mathcal{L}}$ given by

$$P\,\mathcal{S}_0^{\Gamma,\mathcal{L}}\,Q \ \Leftrightarrow_{\mathrm{def}}\ P \in \mathcal{P}^{\Gamma,\mathcal{L}}\ \&\ Q \in \mathcal{P}^{\Gamma,\mathcal{L}}$$

is a $(\Gamma,\mathcal{L})$-bisimulation. Observe also that quasi-strong bisimulation is not preserved by parallel composition, as shown by

$$(\texttt{nil} \parallel y_L := 0) \not\simeq_\Gamma^\mathcal{L} (\texttt{while}\ tt\ \texttt{do}\ \texttt{nil} \parallel y_L := 0)$$

One should notice that, although there exists a largest $(\Gamma,\mathcal{L})$-bisimulation on programs (since the union of a family of such relations is a $(\Gamma,\mathcal{L})$-bisimulation), that we still denote $\approx_\Gamma^\mathcal{L}$, this is not a reflexive relation, since for instance the program $y_L := x_H$ is not bisimilar to itself. However, the relation $\approx_\Gamma^\mathcal{L}$ on programs is a non-empty (as shown by Lemma 3.6) *partial* equivalence relation. The domain of $\approx_\Gamma^\mathcal{L}$ characterizes the *secure* programs:

Definition 3.8 (Secure Programs)
A program P is *secure* in the typing context Γ if and only if $P \approx_\Gamma^\mathcal{L} P$ for any downward-closed set $\mathcal{L}$ of security levels.

We let the reader check that, for instance, the program of Example (1) is not secure in this sense.

To establish the noninterference result, stating that a typable program is secure, we need some preliminary lemmas. The following two lemmas confirm the intuition, discussed earlier, behind the type judgements $\Gamma \vdash P : (\tau, \sigma)\, cmd$. Both results are easily proved by induction on the inference of type judgements.

Lemma 3.9 (Confinement)
If $\Gamma \vdash P : (\tau, \sigma)\, cmd$ then every variable assigned to in P has type $\theta\, var$ in Γ, with $\tau \leq \theta$.

Lemma 3.10 (Guard Safety)
If $\Gamma \vdash P : (\tau, \sigma)\, cmd$ then every loop or conditional guard in P has type θ in Γ, with $\theta \leq \sigma$.

Definition 3.11 ($\mathcal{L}$-Boundedness)
A program P is *$\mathcal{L}$-bounded* in Γ if $\Gamma \vdash P : (\tau, \sigma)\, cmd$ implies $\tau \in \mathcal{L}$.

We shall often use the complementary notion of *non-$\mathcal{L}$-boundedness*: P is *not $\mathcal{L}$-bounded* in Γ if there exist τ, σ, with $\tau \notin \mathcal{L}$, such that $\Gamma \vdash P : (\tau, \sigma)\, cmd$.

Definition 3.12 ($\mathcal{L}$-Guardedness)
A program P is *$\mathcal{L}$-guarded* in Γ if there exist $\sigma \in \mathcal{L}$ and τ such that $\Gamma \vdash P : (\tau, \sigma)\, cmd$.

In the following, we shall most often use the terminology "(non-)$\mathcal{L}$-bounded" and "$\mathcal{L}$-guarded" without any explicit reference to the current typing context Γ. As a consequence of subject reduction, both *non*-$\mathcal{L}$-boundedness and $\mathcal{L}$-guardedness are preserved by execution. Note that, for any downward-closed $\mathcal{L}$, by the Confinement Lemma a program which is *not $\mathcal{L}$-bounded* cannot write on variables of low level, that is in $\mathcal{L}$, and therefore such a program is "high", that is in $\mathcal{P}^{\Gamma,\mathcal{L}}$. Notice however that in general a high program is not necessarily non-$\mathcal{L}$-bounded. For instance, assume that the lattice of security levels is $\{L, A, B, H\}$ with $L = A \sqcap B$ and $H = A \sqcup B$. Then if $\mathcal{L} = \{L\}$, the program $x_A := 0; y_B := 0$ is high and $\mathcal{L}$-bounded in this case. Similarly, by the Guard Safety Lemma, a program which is *$\mathcal{L}$-guarded* does not contain loop guards of high level, that is not in $\mathcal{L}$.

Lemma 3.13 (Behaviour of $\mathcal{L}$-Guarded Programs)
If P is $\mathcal{L}$-guarded in Γ and $\mu =_\Gamma^\mathcal{L} \nu$, then $(P, \mu) \rightarrow (P', \mu')$ implies $(P, \nu) \rightarrow (P', \nu')$, with $\mu' =_\Gamma^\mathcal{L} \nu'$.

Proof: By induction on the inference of $\Gamma \vdash P : (\tau, \sigma)\, cmd$, where $\sigma \in \mathcal{L}$, and by case analysis on the last rule used in this inference. We examine the main cases.

(Assign). Here $P = x := e$ with $\Gamma \vdash e : \tau$, $\Gamma(x) = \tau\, var$, $P' = \texttt{nil}$ and $\mu' = \mu[\mu(e)/x]$. By Assumption 3.2, $\nu(e)$ is defined. Let $\nu' = \nu[\nu(e)/x]$. It is easy to see that $\mu' =_\Gamma^\mathcal{L} \nu'$, since if $\tau \notin \mathcal{L}$ then $\mu' =_\Gamma^\mathcal{L} \nu'$ by the Definition 3.4, while if $\tau \in \mathcal{L}$ then $\mu(e) = \nu(e)$ by Assumption 3.3.

(Seq). Here $P = U; Q$, $\Gamma \vdash U : (\tau_1, \sigma_1)\, cmd$ and $\Gamma \vdash Q : (\tau_2, \sigma_2)\, cmd$ with $\sigma = \sigma_1 \sqcup \sigma_2$ and $\sigma_1 \leq \tau_2$. If the transition $(P, \mu) \rightarrow (P', \mu')$ is proved by means of (Seq-Op1), that is $P' = U'; Q$ with $(U, \mu) \rightarrow (U', \mu')$, then we use the induction hypothesis, since $\sigma_1 \sqcup \sigma_2 \in \mathcal{L} \ \Rightarrow\ \sigma_1 \in \mathcal{L}$. This will give us $(U, \nu) \rightarrow (U', \nu')$, with $\mu' =_\Gamma^\mathcal{L} \nu'$, and by (Seq-Op1) we conclude that $(P, \nu) \rightarrow (U'; Q, \nu')$, as required. In the case where (Seq-Op2) is used, with $U = \texttt{nil}$ and $(Q, \mu) \rightarrow (P', \mu')$, we simply use the induction hypothesis.

(Cond). Here $P = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ with $\Gamma \vdash e : \theta$, $\Gamma \vdash P_i : (\tau, \sigma')\, cmd$, with $\theta \leq \tau$ and $\sigma = \theta \sqcup \sigma'$. Since $\sigma \in \mathcal{L}$, also $\theta \in \mathcal{L}$, and thus $\mu(e) = \nu(e)$ by Assumption 3.3. Hence a transition $(P, \mu) \rightarrow (P_i, \mu)$ is matched by $(P, \nu) \rightarrow (P_i, \nu)$.

(While). Here $P = \texttt{while}\ e\ \texttt{do}\ U$ and $\Gamma \vdash e : \theta$, $\Gamma \vdash U : (\tau, \sigma')\, cmd$, with $\sigma \leq \tau$ and $\sigma = \theta \sqcup \sigma'$. Clearly $\sigma \in \mathcal{L}$ implies $\theta \in \mathcal{L}$, therefore $\mu(e) = \nu(e)$ by Assumption 3.3, and thus a transition of the loop, which is necessarily of the form $(P, \mu) \rightarrow (P', \mu)$, will be matched by $(P, \nu) \rightarrow (P', \nu)$.

The case of (Par) is easy, since the components have the same type as the whole program, and the case of (Subtyping) is immediate by induction. □

This lemma is crucial for the proof of our main results, namely that typable programs (and, in the next Section, scheduled thread systems) are secure. Obviously, it could not be stated in the original system of Volpano, Smith and Irvine [21], where the level of guards is not recorded in the type.

Now, given Γ and $\mathcal{L}$, let us define inductively the relation $\mathcal{S}_1^{\Gamma,\mathcal{L}}$ on sequential programs as follows: $U\, \mathcal{S}_1^{\Gamma,\mathcal{L}}\, V$ if and only if U and V are typable in Γ, and one of the following holds:

1. $U\, \mathcal{S}_0^{\Gamma,\mathcal{L}}\, V$, that is $U \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and $V \in \mathcal{P}^{\Gamma,\mathcal{L}}$, or
2. $V = U$ and U is $\mathcal{L}$-bounded in Γ, or
3. $U = U_0; W$ and $V = V_0; W$, where $U_0\, \mathcal{S}_1^{\Gamma,\mathcal{L}}\, V_0$ and W is not $\mathcal{L}$-bounded in Γ.

Lemma 3.14

(i) *The relation* $\mathcal{S}_1^{\Gamma,\mathcal{L}}$ *is symmetric.*

(ii) *If* U *is typable in* Γ *then* $U\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,U$.

(iii) *If* $\mathtt{nil}\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,V$ *then* $V \in \mathcal{P}^{\Gamma,\mathcal{L}}$.

Proof: The first point is easy to check, by induction on the definition of $\mathcal{S}_1^{\Gamma,\mathcal{L}}$. For the second point, if U is typable in Γ then either U is $\mathcal{L}$-bounded in Γ, in which case $U\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,U$ by clause 2, or not, in which case $U \in \mathcal{P}^{\Gamma,\mathcal{L}}$, as we have seen, hence $U\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,U$ by clause 1. Finally observe that if $\mathtt{nil}\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,V$, this must be by clause 1 or 2, and in both cases this implies $V \in \mathcal{P}^{\Gamma,\mathcal{L}}$. $\square$

Theorem 3.15 (Sequential Noninterference)
The relation $\mathcal{S}_1^{\Gamma,\mathcal{L}}$ *is a quasi-strong* $(\Gamma,\mathcal{L})$*-bisimulation, and therefore if* U *is typable in* Γ *then* U *is secure in* Γ.

Proof: Let

$$\mathcal{R}_1^{\Gamma,\mathcal{L}} = \{((U,\mu),(V,\nu)) \mid U\,\mathcal{S}_1^{\Gamma,\mathcal{L}}\,V \;\&\; \mu =_\Gamma^\mathcal{L} \nu\}$$

By construction, $(U,\mu)\,\mathcal{R}_1^{\Gamma,\mathcal{L}}\,(V,\nu)$ implies $\mu =_\Gamma^\mathcal{L} \nu$. We show, by induction on the definition of $\mathcal{S}_1^{\Gamma,\mathcal{L}}$, that either U and V are both in $\mathcal{P}^{\Gamma,\mathcal{L}}$, or if $(U,\mu) \rightarrow (U',\mu')$ and $(U,\mu)\,\mathcal{R}_1^{\Gamma,\mathcal{L}}\,(V,\nu)$ then there exist V' and ν' such that $(V,\nu) \rightarrow (V',\nu')$, with $(U',\mu')\,\mathcal{R}_1^{\Gamma,\mathcal{L}}\,(V',\nu')$. The case where U and V are both in $\mathcal{P}^{\Gamma,\mathcal{L}}$, that is the case of Clause 1 is trivial, and therefore for the rest of the proof we assume that U and V are not both in $\mathcal{P}^{\Gamma,\mathcal{L}}$. In this proof we implicitly use the Subject Reduction Theorem, which is needed to check that we are dealing with typable terms.

Clause 2: we proceed by induction on the structure of U.

(Assignment). Here $U = x := e$, $U' = \mathtt{nil}$ and $\mu' = \mu[\mu(e)/x]$. Then the matching move is $(U,\nu) \rightarrow (\mathtt{nil},\nu')$, where $\nu' = \nu[\nu(e)/x]$. Indeed from $\mu =_\Gamma^\mathcal{L} \nu$ we deduce $\mu' =_\Gamma^\mathcal{L} \nu'$ since U is $\mathcal{L}$-bounded, and thus by Assumption 3.3 $\mu(e) = \nu(e)$.

(Sequence). Here $U = U_0; W$. We distinguish two cases, according to which of the rules (Seq-Op1) or (Seq-Op2) is used to deduce the transition $(U,\mu) \rightarrow (U',\mu')$

1. If $(U,\mu) \rightarrow (U',\mu')$ is proved by means of (Seq-Op1), that is $U' = U_0'; W$ with $(U_0,\mu) \rightarrow (U_0',\mu')$, then there are two possibilities:

   1.1. W is $\mathcal{L}$-bounded in Γ: then $\Gamma \vdash W : (\tau_2,\sigma_2)\,\mathit{cmd}$ implies $\tau_2 \in \mathcal{L}$, therefore since U is typable there exist τ_1,σ_1 such that $\Gamma \vdash U_0 :$

$(\tau_1, \sigma_1)\, cmd$, with $\sigma_1 \in \mathcal{L}$ (this is where we use the side condition of the rule (Seq)). Thus U_0 is $\mathcal{L}$-guarded. Then by Lemma 3.13 we have $(U_0, \nu) \rightarrow (U'_0, \nu')$, with $\mu' =^{\mathcal{L}}_{\Gamma} \nu'$. Therefore $(U, \nu) \rightarrow (U'_0; W, \nu')$ by (Seq-Op1), which is the required matching move by clause 2 again since $U'_0; W$ is $\mathcal{L}$-bounded (because $\tau_2 \in \mathcal{L}$ implies $\tau_1 \sqcap \tau_2 \in \mathcal{L}$).

1.2. if W is not $\mathcal{L}$-bounded, then $W \in \mathcal{P}^{\Gamma,\mathcal{L}}$. Since we assumed $U \notin \mathcal{P}^{\Gamma,\mathcal{L}}$, U_0 must be $\mathcal{L}$-bounded, and by induction on the structure of the programs, there exist U''_0 and ν' such that $(U_0, \nu) \rightarrow (U''_0, \nu')$ with $U'_0\, \mathcal{S}^{\Gamma,\mathcal{L}}_1\, U''_0$ and $\mu' =^{\mathcal{L}}_{\Gamma} \nu'$. Then $(U, \nu) \rightarrow (U''_0; W, \nu')$ by (Seq-Op1), which is the required matching move by clause 3.

2. Let $(U, \mu) \rightarrow (U', \mu')$ be proved by means of (Seq-Op2), with $U_0 = \texttt{nil}$ and $(W, \mu) \rightarrow (U', \mu')$. Since we assumed $U \notin \mathcal{P}^{\Gamma,\mathcal{L}}$, W must be $\mathcal{L}$-bounded, and by induction on the structure of the programs, there exist U'' and ν' such that $(W, \nu) \rightarrow (U'', \nu')$ with $U'\, \mathcal{S}^{\Gamma,\mathcal{L}}_1\, U''$ and $\mu' =^{\mathcal{L}}_{\Gamma} \nu'$. Then $(U, \nu) \rightarrow (U'', \nu')$ by (Seq-Op2), which is the required matching move.

(Conditional). Here $U = \texttt{if}\ e\ \texttt{then}\ U_1\ \texttt{else}\ U_2$ with $\Gamma \vdash e : \theta$, $\Gamma \vdash U_i : (\tau, \sigma')\, cmd$, with $\theta \leq \tau$ and $\sigma = \theta \sqcup \sigma'$. Then $\tau \in \mathcal{L}$ implies $\theta \in \mathcal{L}$, and by Assumption 3.3 we have $\mu(e) = \nu(e)$. Thus the move $(U, \mu) \rightarrow (U_i, \mu)$ is matched by $(U, \nu) \rightarrow (U_i, \nu)$, since $\mathcal{S}^{\Gamma,\mathcal{L}}_1$ is reflexive. The case of a while loop $U = \texttt{while}\ e\ \texttt{do}\ W$ is similar.

Clause 3: Here $U = U_0; W$ and $V = V_0; W$, where $U_0\, \mathcal{S}^{\Gamma,\mathcal{L}}_1\, V_0$ and W is not $\mathcal{L}$-bounded in Γ. We assumed that U and V are not both in $\mathcal{P}^{\Gamma,\mathcal{L}}$. Then $U_0 \neq \texttt{nil}$, since otherwise we would have $V_0 \in \mathcal{P}^{\Gamma,\mathcal{L}}$ by Lemma 3.14(iii), hence $U \in \mathcal{P}^{\Gamma,\mathcal{L}}$, and $V \in \mathcal{P}^{\Gamma,\mathcal{L}}$, for $W \in \mathcal{P}^{\Gamma,\mathcal{L}}$. Therefore the transition $(U, \mu) \rightarrow (U', \mu')$ must be proved by means of (Seq-Op1), that is $U' = U'_0; W$ with $(U_0, \mu) \rightarrow (U'_0, \mu')$, and we use the induction hypothesis related to $U_0\, \mathcal{S}^{\Gamma,\mathcal{L}}_1\, V_0$ (notice that, due to our assumption about U and V, U_0 and V_0 cannot be both in $\mathcal{P}^{\Gamma,\mathcal{L}}$). □

To extend this result to concurrent programs, we define inductively the relation $\mathcal{S}^{\Gamma,\mathcal{L}}_2$ as follows: $P\, \mathcal{S}^{\Gamma,\mathcal{L}}_2\, Q$ if and only if P and Q are typable in Γ, and one of the following holds:

1. $P\, \mathcal{S}^{\Gamma,\mathcal{L}}_0\, Q$, that is $P \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and $Q \in \mathcal{P}^{\Gamma,\mathcal{L}}$, or
2. $Q = P$ and P is $\mathcal{L}$-bounded in Γ, or
3. $P = U; R$ and $Q = V; R$, where $U\, \mathcal{S}^{\Gamma,\mathcal{L}}_2\, V$ and R is not $\mathcal{L}$-bounded in Γ, or
4. $P = P_1 \parallel P_2$ and $Q = Q_1 \parallel Q_2$ with $P_i\, \mathcal{S}^{\Gamma,\mathcal{L}}_2\, Q_i$.

It is easy to see that Lemma 3.14 holds, replacing sequential programs by concurrent ones, for this relation too.

Theorem 3.16 (Concurrent Noninterference)
The relation $\mathcal{S}_2^{\Gamma,\mathcal{L}}$ is a $(\Gamma,\mathcal{L})$-bisimulation, and therefore if P is typable in Γ then P is secure in Γ.

Proof: For the clauses 1, 2 and 3, the proof is mostly the same as in the previous Theorem (using the proof of Lemma 3.6). The remaining cases only concern parallel composition (in clauses 2 and 4), and are trivial, using the induction hypothesis. □

Observe that if τ_1 and τ_2 are incomparable levels in $(\mathcal{T},\leq)$, they may be separated by a downward-closed set $\mathcal{L}$, namely $\mathcal{L} = \{\tau \mid \tau \leq \tau_1\}$, and therefore a typable program cannot yield any flow from level τ_2 to τ_1 (and conversely). In other words, information flow arising from typable programs is indeed of the kind allowed by the security lattice, that is from τ to σ where $\tau \leq \sigma$.

As we have seen, $(P \parallel Q)$ is typable as soon as both P and Q are. Therefore the previous result means that there can be no information leakage from one secure program to another, concurrent one. Then one may say that our result shows that typed programs are "internally secure", that is they are guaranteed against attacks coming from other, concurrent programs. Notice that, assuming that there is a lowest security level $\bot$, a program that has direct access only to the lowest information is always typable, with type $(\bot,\bot)\ cmd$. Such a program may be regarded as a potential attacker.

As is well-known, bisimulation equivalence is a strong relation, and therefore our noninterference result is quite sharp. In particular, bisimulation equivalence is usually stronger than *trace semantics*, and therefore we can draw some consequences of our result regarding the trace behaviour of programs. To this end, let us introduce a notation: we write $(P,\mu) \rightarrowtail_\Gamma^\mathcal{L} (P',\mu')$ if and only if $\mu' \neq_\Gamma^\mathcal{L} \mu$ and there exist $P_0,\mu_0,\ldots,P_n,\mu_n$ (possibly with $n=0$) such that $\mu_i =_\Gamma^\mathcal{L} \mu$ for all i and

$$(P,\mu) = (P_0,\mu_0) \rightarrow (P_1,\mu_1) \rightarrow \cdots (P_n,\mu_n) \rightarrow (P',\mu')$$

Given an initial memory μ_0, let us define the set, denoted $\mathsf{Trace}_\Gamma^\mathcal{L}(P,\mu_0)$, of $(\mathcal{L},\Gamma)$-*traces* of the program P as the set of finite or infinite sequences $\mu_0,\mu_1,\ldots\mu_n\ldots$ such that there exist $P_1,\ldots P_n\ldots$ with

$$(P,\mu_0) \rightarrowtail_\Gamma^\mathcal{L} (P_1,\mu_1) \rightarrowtail_\Gamma^\mathcal{L} \cdots (P_n,\mu_n) \rightarrowtail_\Gamma^\mathcal{L} \cdots$$

Notice that, by definition, $\mu_{i+1} \neq_\Gamma^\mathcal{L} \mu_i$ for all i. The $(\Gamma,\mathcal{L})$-equality of memories is easily extended into a $(\Gamma,\mathcal{L})$-equality of traces of the same length, as follows:

$$\mu_0,\mu_1,\ldots\mu_n\ldots =_\Gamma^\mathcal{L} \nu_0,\nu_1,\ldots\nu_n\ldots \ \Leftrightarrow_{\mathrm{def}}\ \forall i.\ \mu_i =_\Gamma^\mathcal{L} \nu_i$$

Then we can show that a typable program has the same traces (up to $=^{\mathcal{L}}_{\Gamma}$) when execution starts from $(\Gamma, \mathcal{L})$-equal memories:

Corollary 3.17 (Trace Equivalence)
Let $\mathcal{L}$ be a downward-closed set of security levels. If P is typable in Γ and $\mu =^{\mathcal{L}}_{\Gamma} \nu$ then for any trace $t \in \mathsf{Trace}^{\mathcal{L}}_{\Gamma}(P, \mu)$ there exists $t' \in \mathsf{Trace}^{\mathcal{L}}_{\Gamma}(P, \nu)$ such that $t' =^{\mathcal{L}}_{\Gamma} t$.

Proof: By the Theorem 3.16, we have $(P, \mu) \approx^{\mathcal{L}}_{\Gamma} (P, \nu)$, and it is easy to see that $(P, \mu) \rightarrowtail^{\mathcal{L}}_{\Gamma} (P', \mu')$ and $(P, \mu) \approx^{\mathcal{L}}_{\Gamma} (Q, \nu)$ implies that there exist Q' and ν' such that $(Q, \nu) \rightarrowtail^{\mathcal{L}}_{\Gamma} (Q', \nu')$. □

4 Scheduling Sequential Programs

As pointed out by Smith and Volpano in [18], their noninterference result for concurrent threads relies on the hypothesis of a purely nondeterministic execution. This result breaks down if particular scheduling policies are enforced. We recall the example given in [18]. Assume a *round robin time slicing* scheduler, with a time slice of t steps, $t \geq 2$, and consider the composition $P = U_1 \parallel U_2$ of the following two threads:

$$\begin{array}{rcl} U_1 & = & \texttt{if}\ x_H = 0\ \texttt{then}\ V\ \texttt{else}\ \texttt{nil}\,; \\ & & y_L := 0 \\ U_2 & = & y_L := 1 \end{array} \tag{3}$$

Then, supposing that V is a convergent program that takes at least $t - 1$ steps to execute, and that the scheduler gives precedence to U_1, the value of y_L will depend on that of x_H. The solution proposed in [18] to preserve noninterference in the presence of an arbitrary scheduler consists in forbidding conditionals with high guards([3]), that is, again assuming that there is a lowest security level, to accept only conditional branching on low level expressions. This condition, combined with the exclusion of loops with high guards, required for multi-threading, resulted in [18] in a very severe limitation: the impossibility for any program to test a variable, except at the lowest level. Let us immediately must point out that, with the typing we propose in this paper, we cannot accept *any* scheduling policy. For instance, if the scheduling of two concurrent threads consists in performing the first, up to completion, and then the second, we may reproduce the insecure flow exhibited by the program (1), simply by executing ($\texttt{while}\ x_H = 0\ \texttt{do}\ \texttt{nil} \parallel y_L := 1$) under this scheduling. Therefore, we have

[3] It is also suggested there that a better approach to scheduling would be *probabilistic*. Indeed a whole line of research on probabilistic noninterference has been developed, but this will not be our concern here, where we stick to a *possibilistic* setting.

to reject some scheduling policies, and the way we do that is by encoding these policies into programs, and reject the non typables ones.

One may observe that the program U_1 of Example (3) is actually ruled out by our typing rule for conditional branching, which ensures that low assignments cannot be performed after a high test (*cf.* Example 2). To show that our typing is indeed adequate for dealing with scheduling, we first formalise what it means for a system of concurrent programs to be controlled by a scheduler. Essentially, this means running the system in lockstep with a program that implements the scheduling policy. To describe controlled execution, we use a construction $P[Q]$, which makes P and Q move hand in hand, but allows the controller, P, to proceed by itself whenever Q is unable to move. Then a system consisting of n sequential programs U_i controlled by a scheduler *Sched* will be described as:

$$\mathit{Sched}\ [T_1 \parallel \cdots \parallel T_n]$$

where the T_i are adaptations of the U_i, so that the threads can be triggered and suspended by the scheduler. To this end we introduce a new construct $\texttt{when}\ e\ \texttt{do}\ U$, whose semantics is that U is allowed to proceed, for one step, when the condition e holds. It is technically convenient to introduce two more levels in the syntax: besides the sequential and concurrent programs U and P, written according to the grammar given in Section 2, there is a set of "thread systems" T, and a set of "controlled thread systems" S built as follows:

$$\begin{array}{rcl} T & ::= & \texttt{when}\ e\ \texttt{do}\ U \ \mid\ (T \parallel T) \\ S & ::= & P[T] \end{array}$$

Letting $w(P)$ denote the set of variables written (assigned to) by P, and similarly for $w(T)$, the construct $P[T]$ is only legal under the condition $w(P) \cap w(T) = \emptyset$.

Notation 4.1 *We use $(T, \mu) \rightarrow$ to mean $\exists T', \mu'$ such that $(T, \mu) \rightarrow (T', \mu')$, and $(T, \mu) \not\rightarrow$ for the negation of $(T, \mu) \rightarrow$ (and similarly for the other syntactic categories).*

The semantics of the new constructs is given in Figure 4, where $\mu' \sqcup_\mu \mu''$ represents the memory μ with the conjunction of the updates operated by P and by T, that is $\mu' \backslash \mu \cup \mu'' \backslash \mu \cup (\mu' \cap \mu'')$. As we said, the scheduled sequential programs are written $T_i = \texttt{when}\ e_i\ \texttt{do}\ U_i$ where the expression e_i is the "*proceed*" signal for program U_i, set up by the scheduler. For instance, assuming that each e_i is a variable s_i, initially false, the following program

$$\begin{array}{lll} R_n^t & = & i := 0\,; \\ & & \texttt{while}\ \mathit{tt}\ \texttt{do}\quad k := 0; \\ & & \qquad\qquad (\texttt{while}\ k < t\ \texttt{do}\ s_{i+1} := \mathit{tt}; s_{i+1} := \mathit{ff}; k := k+1); \\ & & \qquad\qquad i := [i+1]_{\mathrm{mod}\ n} \end{array}$$

$$(\textsc{When-Op1}) \quad \frac{\mu(e) = tt,\ (U, \mu) \to (U', \mu')}{(\texttt{when}\ e\ \texttt{do}\ U, \mu) \to (\texttt{when}\ e\ \texttt{do}\ U', \mu')}$$

$$(\textsc{When-Op2}) \quad \frac{\mu(e) = tt,\ (U, \mu) \not\to}{(\texttt{when}\ e\ \texttt{do}\ U, \mu) \to (\texttt{when}\ e\ \texttt{do}\ U, \mu)}$$

$$(\textsc{Control-Op1}) \quad \frac{(P, \mu) \to (P', \mu'),\ (T, \mu) \to (T', \mu'')}{(P[T], \mu) \to (P'[T'], \mu' \sqcup_\mu \mu'')}$$

$$(\textsc{Control-Op2}) \quad \frac{(P, \mu) \to (P', \mu'),\ (T, \mu) \not\to}{(P[T], \mu) \to (P'[T], \mu')}$$

Figure 4: Additional Operational Rules for Systems

---

describes a scheduler for a system of n threads, implementing round robin with time slice t. It is easy to imagine how to program other scheduling policies in a similar style. For instance, assuming that there is a `random` function in the language, such that the evaluation of $\texttt{random}(n)$ consists in randomly choosing an integer i such that $1 \leq i \leq n$, then

$$\texttt{while}\ tt\ \texttt{do}\ (i := \texttt{random}(n); s_i := tt; s_i := f\!f)$$

describes a uniform probabilistic scheduler. Notice also that, if the controlled threads have the form $\texttt{when}\ s \wedge s_i\ \texttt{do}\ U_i$, the scheduling enforced by

$$s_1 := tt; \cdots; s_n := tt; s := tt; \texttt{while}\ tt\ \texttt{do nil}$$

is – assuming that s is initially false – simply the nondeterministic interleaving of the U_i's. We see from these examples that the scheduler usually needs some time to determine the index of the next thread to execute, and this is why we need the rule (Control-Op2). This rule also allows to ignore a terminated thread, and therefore the rule (When-Op2), which is technically convenient, is harmless.

The typing rules for the new operators are given in Figure 5. The side-conditions in rules (When) and (Control) need some comments. First, note that a `when` statement can induce an implicit flow, just like the conditional and `while` statements, as for instance in the system:

$$\texttt{when}\ x_H = 0\ \texttt{do}\ y_L := y_L + 1$$

This explains the requirement $\theta \leq \tau$ in rule (When). Now let us discuss the side condition in the rule (Control). In a controlled thread system $P[T]$

$$\text{(WHEN)} \quad \frac{\Gamma \vdash e : \theta, \quad \Gamma \vdash U : (\tau, \sigma)\, cmd, \quad \theta \leq \tau}{\Gamma \vdash \texttt{when}\ e\ \texttt{do}\ U : (\tau, \theta)\, cmd}$$

$$\text{(CONTROL)} \quad \frac{\Gamma \vdash P : (\tau, \sigma)\, cmd, \quad \Gamma \vdash T : (\tau, \sigma)\, cmd, \quad \tau \geq \sigma}{\Gamma \vdash P[T] : (\tau, \sigma)\, cmd}$$

Figure 5: Additional Typing Rules for Systems

---

where

$$T = (\texttt{when}\ e_1\ \texttt{do}\ U_1 \parallel \cdots \parallel \texttt{when}\ e_n\ \texttt{do}\ U_n)$$

the conditions e_i are normally managed by the controller process P. Then if the U_i's have types $(\tau_i, \sigma_i)\, cmd$, to type $P[T]$ the conditions e_i must be chosen so that they have types θ_i satisfying

$$\theta = \theta_1 \sqcup \cdots \sqcup \theta_n \leq \tau_1 \sqcap \cdots \sqcap \tau_n = \tau$$

If that is the case, $P[T]$ may be typed with $(\tau, \theta)\, cmd$, regardless of the σ_i's, thanks to the side condition in rule (WHEN), and to the fact that we only record the level of the condition as a guard in this rule. Therefore, the side condition $\tau \geq \sigma$ in the rule (CONTROL) is more a constraint for the scheduler P than for the thread system T. Typically, programs of $(L, \_)\, cmd$ type can only be controlled by schedulers of type $(L, L)\, cmd$. One can see for instance that the round robin scheduler R_n^t has type $(L, L)\ cmd$ in a context where the variables i, k and s_i have type $L\ var$. Therefore, for any typable U_i's (in this context([4])), the system

$$R_n^t[\texttt{when}\ s_1\ \texttt{do}\ U_1 \parallel \cdots \parallel \texttt{when}\ s_n\ \texttt{do}\ U_n]$$

is typable. The side condition $\tau \geq \sigma$ in the rule (CONTROL) precludes, for instance, schedulers whose unique type is $(L, H)\, cmd$. Indeed, it would be unsafe to accept programs of this kind as schedulers. For instance, if

$$\begin{aligned} P &= s := tt\, ;\, \texttt{while}\ x_H = 0\ \texttt{do nil} \\ U &= y_L := 0\, ;\, y_L := y_L + 1 \end{aligned}$$

then the system $P[\texttt{when}\ s\ \texttt{do}\ U]$ is not interference-free, and would be typable without the side condition in the rule (CONTROL), with $\Gamma(s) = L\ var$. Let us see another, similar example. Suppose for instance that we wish

[4] Notice that the controlled programs U_i usually do not contain the control variables manipulated by the scheduler. Indeed, i and k should be local to R_n^t – for a treatment of local variables, see [21].

to design a scheduler which is able to detect the termination of a thread U_i, and then to skip it from execution. Then we may transform U_i into $\mathtt{when}\ e_i\ \mathtt{do}\ (U_i\,;t_i := tt)$ where the variable t_i is a termination signal, initially false. Now if U_i has type $(\_, H)\ cmd$, then t_i must be high. Certainly the scheduler will have to test the value of this variable, and therefore it must have type $(\_, H) cmd$, but then, due to the condition $\tau \geq \sigma$ in the (Control) rule, it must have type $(H, H)\,cmd$, and it can only control programs of type $(H, \_)\,cmd$. Indeed, checking the termination of a program with high guards may be dangerous, as shown by the system – with a mistaken, still typable scheduler:

$$P[\mathtt{when}\ s_1\ \mathtt{do}\ (U\ ;\ t_1 := tt) \parallel \mathtt{when}\ s_2\ \mathtt{do}\ (y_L := 0\ ;\ y_L := y_L + 1)]$$

where

$$P \;=\; s_1 := tt\ ;\ s_1 := f\!f\ ;\ s_2 := tt\ ;\ \mathtt{while\ not}\ t_1\ \mathtt{do\ nil}$$

It is easy to check that the Subject Reduction Theorem and the Confinement Lemma, and therefore also Lemma 3.6, extend to the new language. Similarly, the definitions of (strong, quasi-strong) $(\Gamma, \mathcal{L})$-bisimulation and $\mathcal{L}$-boundedness remain formally the same as those for the base language (modulo the replacement of programs by systems), as well as the definition of secure systems. Given a typing context Γ and a downward-closed set $\mathcal{L}$ of security levels, let us define the relation $\mathcal{S}_3^{\Gamma,\mathcal{L}}$ on controlled thread systems as follows: $S_0\,\mathcal{S}_3^{\Gamma,\mathcal{L}}\,S_1$ if and only if S_0 and S_1 are both typable in Γ, and

$$\begin{array}{rcl} S_0 & = & P[\mathtt{when}\ e_1\ \mathtt{do}\ U_1 \parallel \cdots \parallel \mathtt{when}\ e_n\ \mathtt{do}\ U_n] \\ S_1 & = & P[\mathtt{when}\ e_1\ \mathtt{do}\ V_1 \parallel \cdots \parallel \mathtt{when}\ e_n\ \mathtt{do}\ V_n] \end{array}$$

with
1. P is $\mathcal{L}$-guarded in Γ, and
2. for all i there exists $\theta_i \in \mathcal{L}$ such that $\Gamma \vdash e_i : \theta_i$, and
3. $U_i \simeq_\Gamma^\mathcal{L} V_i$ for all i.

Lemma 4.2 *The relation $\mathcal{S}_3^{\Gamma,\mathcal{L}}$ is a strong $(\Gamma, \mathcal{L})$-bisimulation.*

Proof: Let $\mu =_\Gamma^\mathcal{L} \nu$ and $(S_0, \mu) \rightarrow (S_0', \mu')$. There are two cases:

1. If this transition is proved by means of (Control-Op1) then

$$S_0' \;=\; P'[\cdots \parallel \mathtt{when}\ e_i\ \mathtt{do}\ U_i' \parallel \cdots]$$

where $(P, \mu) \rightarrow (P', \mu_0)$, $(\mathtt{when}\ e_i\ \mathtt{do}\ U_i, \mu) \rightarrow (\mathtt{when}\ e_i\ \mathtt{do}\ U_i', \mu_1)$ and $\mu' = \mu_0 \sqcup_\mu \mu_1$. Since P is $\mathcal{L}$-guarded in Γ, by Lemma 3.13, there exists ν_0 such that $(P, \nu) \rightarrow (P', \nu_0)$ with $\nu_0 =_\Gamma^\mathcal{L} \mu_0$. Regarding the transition of the thread system, there are again two cases:

1.1. The transition is proved by means of (When-Op1), that is $\mu(e) = tt$ and $(U_i, \mu) \rightarrow (U_i', \mu_1)$. Since $V_i \simeq_\Gamma^\mathcal{L} U_i$, either there is a transition $(V_i, \nu) \rightarrow (V_i', \nu_1)$ such that $V_i' \simeq_\Gamma^\mathcal{L} U_i'$ and $\nu_1 =_\Gamma^\mathcal{L} \mu_1$, or both U_i and V_i are in $\mathcal{P}^{\Gamma,\mathcal{L}}$. By the Assumption 3.3 we have $\nu(e_i) = \mu(e_i) = tt$ (since e_i is low in Γ), and therefore in the first case $(\texttt{when}\ e_i\ \texttt{do}\ V_i, \nu) \rightarrow (\texttt{when}\ e_i\ \texttt{do}\ V_i', \nu_1)$ by (When-Op1), hence

$$(S_1, \nu) \rightarrow (P'[\cdots \parallel \texttt{when}\ e_i\ \texttt{do}\ V_i' \parallel \cdots], \nu')$$

where $\nu' = \nu_0 \sqcup_\nu \nu_1 =_\Gamma^\mathcal{L} \mu'$, and this is the required matching move. Otherwise, $U_i \in \mathcal{P}^{\Gamma,\mathcal{L}}$ and $V_i \in \mathcal{P}^{\Gamma,\mathcal{L}}$, and we distinguish again two cases: either there exist V_i' and ν_1 such that $(V_i, \nu) \rightarrow (V_i', \nu_1)$, and then $\nu_1 =_\Gamma^\mathcal{L} \nu =_\Gamma^\mathcal{L} \mu =_\Gamma^\mathcal{L} \mu_1$, and we conclude as in the previous case, or $(V_i, \nu) \not\rightarrow$. Then $(\texttt{when}\ e_i\ \texttt{do}\ V_i, \nu) \rightarrow (\texttt{when}\ e_i\ \texttt{do}\ V_i, \nu)$ by (When-Op2), with $\nu =_\Gamma^\mathcal{L} \mu =_\Gamma^\mathcal{L} \mu_1$, and

$$(S_1, \nu) \rightarrow (P'[\cdots \parallel \texttt{when}\ e_i\ \texttt{do}\ V_i \parallel \cdots], \nu_0)$$

is the required matching move in this case.

1.2. The transition is proved by means of (When-Op2), that is $\mu(e) = tt$, $(U_i, \mu) \not\rightarrow$ and $U_i' = U_i$ and $\mu_1 = \mu$. Since $V_i \simeq_\Gamma^\mathcal{L} U_i$, we have $V_i \in \mathcal{P}^{\Gamma,\mathcal{L}}$, and we distinguish two cases, depending on whether (V_i, ν) may perform a transition or not. We conclude as in the previous case.

2. If this transition is proved by means of (Control-Op2) then $S_0' = P'[T_0]$ with $(P, \mu) \rightarrow (P', \mu')$ and $(T_0, \mu) \not\rightarrow$, where

$$T_0 \;=\; (\texttt{when}\ e_1\ \texttt{do}\ U_1 \parallel \cdots \parallel \texttt{when}\ e_n\ \texttt{do}\ U_n)$$

It is easy to see that $(T_0, \mu) \not\rightarrow$ implies $\mu(e_i) = \mathit{ff}$ for all i, and therefore also $\nu(e_i) = \mathit{ff}$ by Assumption 3.3 since the e_i's are low in Γ. Then $(T_1, \nu) \not\rightarrow$ where

$$T_1 \;=\; (\texttt{when}\ e_1\ \texttt{do}\ V_1 \parallel \cdots \parallel \texttt{when}\ e_n\ \texttt{do}\ V_n)$$

Since P is $\mathcal{L}$-guarded in Γ, by Lemma 3.13, there exists ν' such that $(P, \nu) \rightarrow (P', \nu')$ with $\nu' =_\Gamma^\mathcal{L} \mu'$, and therefore $(S_1, \nu) \rightarrow (P'[T_1], \nu')$, and this is the required matching move. $\square$

We are now ready to prove our main result:

Theorem 4.3 (Noninterference for Controlled Thread Systems)
If S is a controlled thread system typable in Γ then S is secure in Γ.

Proof: Given a downward-closed set $\mathcal{L}$ of security levels, let us still denote by $\mathcal{P}^{\Gamma,\mathcal{L}}$ the set of high programs and systems (of any kind), and by $\mathcal{S}_0^{\Gamma,\mathcal{L}}$

the cartesian product $\mathcal{P}^{\Gamma,\mathcal{L}} \times \mathcal{P}^{\Gamma,\mathcal{L}}$. It is easy to see (as in Lemma 3.6) that this is a $(\Gamma,\mathcal{L})$-bisimulation, and therefore the relation

$$\mathcal{S}_4^{\Gamma,\mathcal{L}} \;=\; \mathcal{S}_0^{\Gamma,\mathcal{L}} \cup \mathcal{S}_3^{\Gamma,\mathcal{L}}$$

is a (quasi-strong) $(\Gamma,\mathcal{L})$-bisimulation, by the previous lemma. We show that if S is typed in Γ then $S\,\mathcal{S}_4^{\Gamma,\mathcal{L}}\,S$. This is obvious if S is not $\mathcal{L}$-bounded in Γ, by the (extended) Confinement Lemma, since S is in $\mathcal{P}^{\Gamma,\mathcal{L}}$ in this case. If S is $\mathcal{L}$-bounded in Γ, and $S = P[T]$ where

$$T \;=\; (\texttt{when}\ e_1\ \texttt{do}\ U_1 \parallel \cdots \parallel \texttt{when}\ e_n\ \texttt{do}\ U_n)$$

then a typing $\Gamma \vdash S : (\tau,\sigma)\,cmd$ may be inferred either using the (Subtyping) or the (Control) rule. More precisely, there exist τ' and σ' such that $\Gamma \vdash S : (\tau',\sigma')\,cmd$ by the (Control) rule, that is $\Gamma \vdash P : (\tau',\sigma')\,cmd$ and $\Gamma \vdash T : (\tau',\sigma')\,cmd$ with $\tau \leq \tau' \geq \sigma'$ and $\sigma' \leq \sigma$. Since S is $\mathcal{L}$-bounded in Γ, we have $\sigma' \in \mathcal{L}$, and therefore P is $\mathcal{L}$-guarded in Γ. Now $\Gamma \vdash T : (\tau',\sigma')\,cmd$ may be proved either using the (Subtyping) or the (Par) rule – or the (When) rule if $n = 1$. Then it is not difficult to see that there exist θ_i, τ_i and σ_i for each i such that $\Gamma \vdash e_i : \theta_i$ and $\Gamma \vdash U_i : (\tau_i,\sigma_i)\,cmd$ with $\tau' \leq \tau_1 \sqcap \cdots \sqcap \tau_n$ and $\theta_1 \sqcup \cdots \sqcup \theta_n \leq \sigma'$ and therefore $\theta_i \in \mathcal{L}$ for all i. Finally we have $U_i \simeq_\Gamma^\mathcal{L} U_i$ for all i by Theorem 3.15, and this concludes the proof. □

5 Conclusion and related work

We have addressed the question of secure information flow in systems of concurrent programs. This covers one of the security problems that can arise, for instance, when a mobile program visits different sites, namely that of preserving the *confidentiality* of the visited sites' private data. In fact, in [5], it is shown how a form of noninterference called *non deducibility on composition* may be used to model also other security properties like *authenticity*, *non repudiation* and *fairness*. Noninterference thus appears as a rather interesting notion to study when security is concerned. On the other hand, it may be argued [10] that *covert channels*, that is implicit information flows, are unavoidable in practice, as they can arise also at the hardware level. Thus the aim of statically ensuring the absence of covert channels might be a hard one to realise. We certainly do not claim here to cover the whole range of possible attacks from a hostile party.

The issue of noninterference has been largely studied in the literature, using different models, and it is not our intention to review the various approaches. We focussed on the approach of Volpano, Smith et al., as it gives an elegant treatment for a fairly standard language, which can be assumed to be the kernel of more sophisticated practical languages.

The question of secure information flow and noninterference has also been investigated in the setting of process calculi [5, 15]. More recently, there have been studies on noninterference for functional languages [14] and mobile process calculi [9], [8] and [7]. The latter papers are closer to our work, as they use a type system to enforce noninterference. The treatment in [9] and [8], however, seems overly restrictive: it amounts (at least in the core calculus) to forbid all control flow from actions on high channels to actions on low channels. In [9], the core calculus is extended with more sophisticated constructs; in the extended calculus some actions may be classified as "innocuous", and the restriction on control flow may be relaxed when these actions are involved. The papers [14] and [7] are less restrictive and closer in spirit to our approach, as they try to distinguish the dangerous control flow (implementing information flow) from the harmless control flow which should not be restricted. Another related paper is [1], which studies secrecy properties in security protocols expressed in the *spi*-calculus. There again, a type system is used to ensure the security properties.

As concerns noninterference in the presence of scheduling policies, the most popular approach has been so far the probabilistic one, taken for instance in [20] and [16]. Our intention here was to handle scheduling entirely within a possibilistic setting. It should be noted, also, that in [20] and [16] scheduling is introduced at the semantic level (adding probabilities to the transitions), while we express scheduling policies – which, after all, are programs – at the syntactic level. This has the advantage of allowing us to type the schedulers themselves, and express through typing conditions the restrictions one needs to impose on any reasonable scheduling policy for multilevel systems. This is in line with the view expressed by Sabelfeld and Sands in [16], where a scheduler is described as a "mechanism for selecting threads which itself satisfies some noninterference property". Moreover, [16] also identifies a few characteristics which are deemed necessary for schedulers to be well-behaved in a multilevel security setting. Essentially, schedulers shouldbe allowed to have access to the low part of the memory (to allow some interaction with the controlled threads), and be able to use the *history* of thread creation/generation along the computation, which may be of high level. In our setting we are able to express more precise conditions, for instance that schedulers allowed to test the termination of high threads should themselves be high level programs. However, it would be interesting to compare more precisely our syntactic approach with the probabilistic approach of [20, 16].

As we have seen, conditional branchings combined with scheduling can introduce new interferences due to the unbalanced duration of the branches. Some proposed solutions to this problem rely on a controlled execution time for conditional branchings with high guards. For instance, in [20], a "protect" construct is used to encapsulate these conditional branchings as atomic operations. Another solution was presented by Agat [2], and proved correct in [16], where the branches of high-guarded conditionals are padded so that they always have the same duration. It should be noted that both these approaches allow loops of low level only. Here we have adopted a different solution to this specific problem, which allows us to deal uniformly with various notions of guards. For instance, with our notion of type it is very easy to deal with the standard `wait/signal` primitives used for synchronisation and cooperative scheduling.

By the time we started revising this paper, and thanks also to one of the anonymous referees, we became aware of an independent paper by G. Smith [17] which was about to appear (indeed, within a similar time frame as our extended abstract [4]). As it turns out, Smith's paper proposes a type system which is identical to ours, with the additional possibility of recording the running time of programs when this is known statically, thus incorporating the above mentioned timing techniques for conditional branchings in the type system. Indeed, Smith's concern is in controlling *time leaks*, of which scheduling leaks are a particular case. The absence of time leaks is formalised as a property of *probabilistic noninterference*. Since our type system is slightly more restrictive than Smith's one (in the sense that the class of typable programs is smaller), it also implies a form of probabilistic noninterference.

Finally, let us mention that the type system presented here is different from the one of [4], where the level of conditional guards does not need to be taken into account as long as scheduling is not considered. Instead the typing rule for loops is more restricted: loops are only allowed to have uniform types of the form $(\tau, \tau)cmd$. This alternative type system also guarantees noninterference for concurrent programs (whithout scheduling), and we conjecture that it allows a larger class of programs to be typed. On the other hand, the solution for scheduling we propose here is more generous than that of [4], as long as we are only interested in scheduling sequential threads.

An issue which has not been addressed here, but is planned for future work, is the feasibility of checking noninterference using a type inference algorithm, in the line of [19]. Current work is also oriented towards the treatment of more realistic languages, as advocated for instance in [12], including exceptions and higher-order constructs.

Acknowledgements

We would like to thank Geoffrey Smith for his interest in, and comments on the submitted version of this paper, and the anonymous referees, whose questions and suggestions led us to largely revise and improve the last section of the paper.

References

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] J. Agat. Transforming out timing leaks. In *27th ACM Symposium on Principles of Programming Languages (POPL)*, pages 40–53, 2000.

[3] P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J.-L. Lanet. Electronic purse applet certification (extended abstract). In *Workshop on Secure Architectures and Information Flow*, number 32 in ENTCS, 2000.

[4] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP'01*, number 2076 in LNCS, pages 382–395, 2001.

[5] R. Focardi, R. Gorrieri, and F. Martinelli. Non interference for the analysis of cryptographic protocols. In *Proceedings ICALP'00*, number 1853 in LNCS, 2000.

[6] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[7] M. Hennessy. The security π-calculus and noninterference. Computer Science Technical Report 2000:05, University of Sussex, 2000.

[8] M. Hennessy and J. Riely. Information flow vs resource access in the asynchronous pi-calculus (extended abstract). In *Proceedings ICALP'00*, number 1853 in LNCS, 2000.

[9] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proceedings ESOP'00*, number 1782 in LNCS, pages 180–199, 2000.

[10] J. Millen. 20 years of covert channel modeling and analysis. In *IEEE Symposium on Security and Privacy*, 1999.

[11] R. Milner. *A Calculus of Communicating Systems*, volume 92 of LNCS. Springer-Verlag, 1980.

[12] A. Myers. Jflow: Practical mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL)*, 1999.

[13] D. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI-Conf. on Theoretical Computer Science*, number 104 in LNCS, 1981.

[14] F. Pottier and S. Conchon. Information flow inference for free. In *Proceedings ICFP'00*, 2000.

[15] P.Y.A. Ryan and S.A. Schneider. Process algebra and non-interference. *Journal of Computer Security*, 9(1/2), 2001.

[16] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In IEEE, editor, *13th Computer Security Foundations Workshop*, 2000.

[17] G. Smith. A new type system for secure information flow. In *14th IEEE Computer Security Foundations Workshop*, 2001.

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

[19] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97*, number 1214 in LNCS, pages 607–621, 1997.

[20] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3), 1999.

[21] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.