Noninterference for Concurrent Programs*

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 extended with parallelism. Our proposal is in the line of some recent work by Irvine, Volpano and Smith. Our type system, as well as our semantics for concurrent programs, seem more natural and less restrictive than those originally presented by these authors. Moreover, we show how to adapt the type system in order to preserve the noninterference results in the presence of scheduling policies, while remaining in a nonprobabilistic setting.

1 Introduction

The aim of this paper is to study the notion of secure information flow, and more specifically of *noninterference* (a notion first introduced by Goguen and Meseguer in [4]) in the setting of concurrency. Our starting point is the paper [15] by Volpano, Smith and Irvine, and the subsequent paper [12] 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 [15] is purely sequential, and is extended in [12] with asynchronous parallelism (interleaving). In this introduction, and in the examples given in the paper, the security levels will simply be high and low. High-level variables are supposed to contain *secret* information, while low-level variables contain *public* information. However all results will be given for an arbitrary lattice of security levels.

In Volpano et al.'s work, *noninterference* means that variables of a given level do not interfere with those of lower levels: more precisely, the values of low-level variables are not dependent on the values of high-level variables. Noninterference is meant to model the absence of information flow from high level to low level. Such information flow is considered *insecure*, as it amounts to the disclosure of secret information into the public domain. 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, for instance. In the approach of [15,12], these situations are prevented by means of a type system. More precisely, 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)

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

F. Orejas, P.G. Spirakis, and J. van Leeuwen (Eds.): ICALP 2001, LNCS 2076, pp. 382–395, 2001.

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

PIN, t_α, t_β : boolean variables of type H

r : boolean variable of type L

γ : thread of type H, α, β : threads of type L

Fig. 1. Information Flow through Control Flow

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, 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 Volpano and Smith's type system, 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 [12]. 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 $t_\alpha = t_\beta = ff$ the effect of the program is to copy the value of the secret variable *PIN* into the public variable r. The illicit information flow from *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, this particular system is to be accepted.

To circumvent this problem, Smith and Volpano propose in [12] to forbid the use of high-level variables as guards in while-loops, that is, assuming that there is a lowest security level, 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. Technically, this means that we will abandon the *big-step semantics* which is the basis of Volpano et al.'s analysis in favor of a *small step semantics* for programs, which is the approach usually adopted in dealing with parallelism. 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 type system, where the level of a guard – the expression tested by a `while` loop – is taken into account in sequential composition.

We will also examine the situation where a scheduling policy is in force in a thread system: we will introduce a few new programming primitives to describe formally such a situation, and show how to adapt the type system for this new setting, where new interference phenomena arise. As can be expected, this will result in a slight restriction on the type of certain programs, though not as severe as that prefigured in [12].

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. The proofs are omitted from this extended abstract. They are to be found in the full version of the paper [2].

2 The Language and Type System

The language we consider is essentially that of [12] (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:

$$U,\ V \ldots := \texttt{nil} \mid x := e \mid U; V \mid \texttt{if } e \texttt{ then } U \texttt{ else } V \mid \texttt{while } e \texttt{ do } U$$

$$P,\ Q \ldots := U \mid U; P \mid \texttt{if } e \texttt{ then } P \texttt{ else } Q \mid \texttt{while } e \texttt{ do } P \mid P \parallel Q$$

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 [12], which describes concurrent systems as collections of threads, thus allowing only top-level parallelism, while we allow the dynamic spawning of new threads.

The operational semantics of the language is given in terms of transitions between configurations $(P, \mu) \rightarrow (P', \mu')$ 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 [12]. We use the notation $\mu[v/x]$ for memory update. The rules specifying the operational semantics of programs are presented in Figure 2. As pointed out already in the introduction, the semantics used here is a *small step semantics*, as opposed to the *big step semantics* of [12][1]. The rules are fairly standard, and we shall not comment on them.

In the introduction we argued that, in a small-steps semantics, 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 [15] – however as suggested by the above example it will be more restrictive than that of [15] on the sequential sublanguage, because of our more detailed observation of programs.

The types of data and expressions are *security levels*, that is elements of a lattice $(\mathcal{S}, \leq)$. We denote the operations of meet and join respectively by $\sqcap$ and $\sqcup$. These types are ranged over by τ, σ. 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 [15] concerns the types for programs. Type judgements in [15] are of the form $\Gamma \vdash P : \tau\, cmd$, where Γ is a mapping from variables to types of variables, i.e. elements of $\{\tau\, var \mid \tau \in \mathcal{S}\}$. The meaning of $\Gamma \vdash P : \tau\, cmd$ is that in the type environment Γ, 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.

To take into account loop guards, 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 guards occurring in a program. Accordingly, the subtyping for programs is contravariant in its first component and covariant in the second:

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

[1] In fact, the semantics of [12] 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.

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

$$\text{(Seq-Op1)}\quad \frac{(U, \mu) \to (U^o, \mu^o)}{(U; P, \mu) \to (U^o; P, \mu^o)}$$

$$\text{(Seq-Op2)}\quad \frac{(P, \mu) \to (P^o, \mu^o)}{(\mathtt{nil}; P, \mu) \to (P^o, \mu^o)}$$

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

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

$$\text{(While-Op1)}\quad \frac{\mu(e) = tt}{(\mathtt{while}\ e\ \mathtt{do}\ P, \mu) \to (P; \mathtt{while}\ e\ \mathtt{do}\ P, \mu)}$$

$$\text{(While-Op2)}\quad \frac{\mu(e) \neq tt}{(\mathtt{while}\ e\ \mathtt{do}\ P, \mu) \to (\mathtt{nil}, \mu)}$$

$$\text{(Par-Op1)}\quad \frac{(P, \mu) \to (P^o, \mu^o)}{(P \parallel Q, \mu) \to (P^o \parallel Q, \mu^o)}$$

$$\text{(Par-Op2)}\quad \frac{(Q, \mu) \to (Q^o, \mu^o)}{(P \parallel Q, \mu) \to (P \parallel Q^o, \mu^o)}$$

Fig. 2. Operational Semantics for Parallel Programs

The guard type will be set up by while-loops and looked up by sequential composition. The complete type system for programs is shown in Figure 3. Notice that the guard type plays no particular rôle in rules (Nil), (Assign) and (Cond), which are plain adaptations of the ones in [15]. Let us comment a little on the rules for while-loops and sequential composition, which are the main novelty w.r.t. [15,12]. As explained, the guard type is σ for a while-loop testing an expression of level σ, and from then onwards it should stay equal to σ to prevent concatenation with low-level programs. 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). One may notice that rule (While) imposes types of the form $(\tau, \tau)\, cmd$ to while-loops (by subtyping they also have types $(\theta, \sigma)\, cmd$ with $\theta \leq \sigma$). We let the reader check that, had we accepted for instance $(H, L)\, cmd$, we would not avoid interferences, as shown by the example

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

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

$$\text{(Seq)}\quad \frac{\Gamma \vdash U : (\tau, \sigma)\ cmd, \quad \Gamma \vdash P : (\tau^0, \sigma^0)\ cmd, \quad \sigma \leq \tau^0}{\Gamma \vdash U\,;\, P : (\tau \sqcap \tau^0, \sigma \sqcup \sigma^0)\ cmd}$$

$$\text{(Cond)}\quad \frac{\Gamma \vdash e : \tau, \quad \Gamma \vdash P : (\tau, \sigma)\ cmd, \quad \Gamma \vdash Q : (\tau, \sigma)\ cmd}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q : (\tau, \sigma)\ cmd}$$

$$\text{(While)}\quad \frac{\Gamma \vdash e : \tau, \quad \Gamma \vdash P : (\tau, \tau)\ cmd}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ P : (\tau, \tau)\ cmd}$$

$$\text{(Par)}\quad \frac{\Gamma \vdash P : (\tau, \sigma)\ cmd, \quad \Gamma \vdash Q : (\tau, \sigma)\ cmd}{\Gamma \vdash P \parallel Q : (\tau, \sigma)\ cmd}$$

$$\text{(Subtyping)}\quad \frac{\Gamma \vdash P : (\tau, \sigma)\ cmd, \quad \tau^0 \leq \tau, \quad \sigma \leq \sigma^0}{\Gamma \vdash P : (\tau^0, \sigma^0)\ cmd}$$

Fig. 3. Typing Rules for Concurrent Programs

$$\begin{array}{l} \texttt{if}\ \ x_H = 0 \quad \texttt{then}\ \texttt{while}\ \ y_L = 0\ \ \texttt{do}\ \texttt{nil} \\ \qquad\qquad\quad\ \ \texttt{else}\ \texttt{nil}\,; \\ u_L := u_L + 1 \end{array} \tag{2}$$

Similarly, we have to rule out the insecure program

$$\texttt{while}\ \ tt\ \ \texttt{do}\ \ (\, y_L := y_L + 1\ ;\ \texttt{while}\ \ x_H = 0\ \ \texttt{do}\ \texttt{nil}\,) \tag{3}$$

and this shows why loops having $(L, H)\ cmd$ as their unique type should be forbidden.

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. □

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.

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, for any type environment Γ and security level ω, a notion of equality on memories which formalises the idea that two memories coincide on variables of level less than or equal to ω in Γ. Intuitively, such memories are indistinguishable for an observer of level ω.

Definition 3.4 (ω-Equality of Memories)
$\mu =^{\omega}_{\Gamma} \nu \;\Leftrightarrow_{\rm def}\; \forall x.\, \Gamma(x) = \tau\, var \;\&\; \tau \leq \omega \;\Rightarrow\; \mu(x) = \nu(x)$.

Definition 3.5 ((Γ,ω)-Bisimulation)
A relation $\mathcal{R}$ on configurations is a (Γ,ω)-bisimulation if $(P,\mu)\,\mathcal{R}\,(Q,\nu)$ implies

(i) $\mu =^{\omega}_{\Gamma} \nu$
(ii) $(P,\mu) \to (P',\mu') \;\Rightarrow\; \exists Q', \nu'.\, (Q,\nu) \to^* (Q',\nu') \;\wedge\; (P',\mu')\,\mathcal{R}\,(Q',\nu')$
(iii) $(Q,\nu) \to (Q',\nu') \;\Rightarrow\; \exists P', \mu'.\, (P,\mu) \to^* (P',\mu') \;\wedge\; (P',\mu')\,\mathcal{R}\,(Q',\nu')$

The (Γ,ω)-*bisimulation equivalence* on configurations, noted $\approx^{\omega}_{\Gamma}$, is the largest (Γ,ω)-bisimulation.

The following two lemmas confirm the intuition, discussed earlier, behind the type judgements $\Gamma \vdash P : (\tau,\sigma)\,cmd$.

Lemma 3.6 (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 3.7 (Guard Safety)
If $\Gamma \vdash P : (\tau,\sigma)\,cmd$ then every loop guard in P has type θ in Γ, with $\theta \leq \sigma$.

Definition 3.8 (ω-Boundedness)
A program P is ω-*bounded* if $\Gamma \vdash P : (\tau,\sigma)\,cmd$ implies $\tau \leq \omega$.

Definition 3.9 (ω-Guardedness)
A program P is ω-*guarded* if there exist τ,σ, with $\sigma \leq \omega$, such that $\Gamma \vdash P : (\tau,\sigma)\,cmd$.

Note that by the Confinement Lemma, a program which is *not* ω-*bounded* cannot write on variables of level less than or equal to ω. Similarly, by the Guard Safety Lemma, a program which is ω-*guarded* does not contain loop guards of level higher than or incomparable with ω. As a consequence of subject reduction, both *non*-ω-boundedness and ω-guardedness are preserved by execution.

Proposition 3.10 (Bisimilarity of Non ω-Bounded Programs)
Let $\mathcal{S}^{\Gamma,\omega}$ be the relation consisting of the pairs $((P,\mu),(Q,\nu))$ such that $\mu =^{\omega}_{\Gamma} \nu$ and there exist τ, σ and τ', σ' with $\tau \not\leq \omega$, $\tau' \not\leq \omega$, such that $\Gamma \vdash P : (\tau,\sigma)\,cmd$ and $\Gamma \vdash Q : (\tau',\sigma')\,cmd$. Then $\mathcal{S}^{\Gamma,\omega}$ is a (Γ,ω)-bisimulation.

Proof: Let $((P,\mu),(Q,\nu)) \in \mathcal{S}^{\Gamma,\omega}$ and $(P,\mu) \to (P',\mu')$. This can be matched by $(Q,\nu) \to^* (Q,\nu)$, since by the Confinement Lemma $\mu' =^{\omega}_{\Gamma} \mu =^{\omega}_{\Gamma} \nu$, and by the Subject Reduction Theorem $\Gamma \vdash P' : (\tau,\sigma)$. □

Note that (Γ,ω)-bisimilarity does not preserve termination. For instance, for any memories μ and ν such that $\mu =^{\omega}_{\Gamma} \nu$ we have:

$$(\,\texttt{nil},\,\mu\,) \;\approx^{\omega}_{\Gamma}\; (\,\texttt{while}\ tt\ \texttt{do nil},\,\nu\,)$$

We introduce now a notion which will play a key rôle for noninterference.

Definition 3.11 (ω-Constrainment)
A program P is *ω-constrained* if there exist τ,σ, with $\tau \not\leq \omega$ and $\sigma \leq \omega$, such that $\Gamma \vdash P : (\tau,\sigma)\,cmd$.

By definition any ω-constrained program is both ω-guarded and *not* ω-bounded. It is worth stressing that the converse is not true, as shown by the program `while` tt `do nil`. Clearly, for any type environment Γ, this program is ω-guarded for any security level ω and not ω-bounded if $\omega \neq \top$. However it is not ω-constrained, as a consequence of the uniform typing in rule (While). Indeed, an important property of ω-constrained programs is the following.

Lemma 3.12 (Termination of ω-Constrained Sequential Programs)
If U is ω-constrained, then for any μ there exist μ', U' such that $(U,\mu) \to^* (U',\mu')$ and $U' = \texttt{nil};\cdots;\texttt{nil}$.

Finally we can state our main result:

Theorem 3.13. (Noninterference)
If P is typable in Γ, then $(P,\mu) \approx^{\omega}_{\Gamma} (P,\nu)$ for any μ,ν such that $\mu =^{\omega}_{\Gamma} \nu$.

Proof: We define inductively the relation $\mathcal{R}_0^{\Gamma,\omega}$ on configurations as follows: $(P,\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(Q,\nu)$ if and only if P and Q are typable, $\mu =^{\omega}_{\Gamma} \nu$ and one of the following holds:

1. $(P,\mu) \approx^{\omega}_{\Gamma} (Q,\nu)$
2. $P = Q$ and P is ω-bounded
3. $P = U;R$ and $Q = V;R$, where both U and V are ω-constrained
4. $P = U;R$ and $Q = V;R$, where $(U,\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(V,\nu)$ and R is not ω-bounded
5. P is not ω-bounded and $Q = V;R$, where $(\texttt{nil},\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(V,\nu)$ and R is not ω-bounded (or symmetrically)
6. $P = P_1 \parallel P_2$ and $Q = Q_1 \parallel Q_2$ with $(P_i,\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(Q_i,\nu)$.

We show that $\mathcal{R}_0^{\Gamma,\omega}$ is a (Γ,ω)-bisimulation. The theorem will be a consequence of this fact, since if P is typable, then either P is not ω-bounded, in which case $(P,\mu) \approx^{\omega}_{\Gamma} (P,\nu)$ by Proposition 3.10, or P is ω-bounded and $(P,\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(P,\nu)$ by the second clause of the definition. In the case of Clause 3, we use the Lemma 3.12. □

4 Adding a Scheduler

As pointed out by Smith and Volpano in [12], noninterference results such as those of the previous section rely on the hypothesis of a purely nondeterministic execution of concurrent programs. These results would break down if particular scheduling policies were enforced. We recall the example given in [12]. Assume a *round robin time slicing* scheduler, with a time slice of t steps, $t \geq 2$, and consider the composition $P = \alpha \parallel \beta$ of the following two threads:

$$\begin{aligned} \alpha : & \texttt{if } x_H = 0 \texttt{ then } Q \texttt{ else nil}; \\ & y_L := 0 \\ \beta : & \, y_L := 1 \end{aligned} \tag{4}$$

Then, supposing that Q is a convergent program that takes at least $t-1$ steps to execute, and that the scheduler gives precedence to α, the value of y_L will depend on that of x_H. The solution proposed in [12] to preserve noninterference in the presence of an arbitrary scheduler consists in forbidding conditionals with high guards[2], 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 [12] in a very severe limitation: the impossibility for any program to test a variable, except at the lowest level.

We present here a different solution for scheduling, which does not rule out conditionals with high guards. To this end 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 parallel programs P_i controlled by a scheduler *Sched* will be described as:

$$Sched\ [P'_1 \parallel \cdots \parallel P'_n]$$

where the P'_i are adaptations of the P_i, so that the processes can be triggered and suspended by the scheduler. To this end we introduce a new construct $\texttt{when } e \texttt{ do } P$, whose semantics is that P is allowed to proceed, for one step, when the condition e holds. It is technically convenient to introduce another level in the syntax: besides the programs P, written according to the grammar given in Section 2, there is a set of "systems" S, T built as follows:

$$S \ ::= \ P \ \mid \ S \parallel T \ \mid \ S[T] \ \mid \ \texttt{when } e \texttt{ do } S$$

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

[2] 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.

$$\text{(Control-Op1)}\quad \frac{(S,\mu)\to(S^{o},\mu^{o}),\ (T,\mu)\to(T^{o},\mu^{o})}{(S[T],\ \mu)\to(S^{o}[T^{o}],\mu^{o}\sqcup_{\mu}\mu^{o})}$$

$$\text{(Control-Op2)}\quad \frac{(S,\mu)\to(S^{o},\mu^{o}),\quad (T,\mu)\not\to}{(S[T],\mu)\to(S^{o}[T],\mu^{o})}$$

$$\text{(When-Op)}\quad \frac{\mu(e)=tt,\qquad (S,\mu)\to(S^{o},\mu^{o})}{(\texttt{when}\ e\ \texttt{do}\ S,\ \mu)\to(\texttt{when}\ e\ \texttt{do}\ S^{o},\ \mu^{o})}$$

Fig. 4. Additional Operational Rules for Systems

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

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

Fig. 5. Additional Typing Rules for Systems

Notation 4.1 *We use* $(S,\mu)\to$ *to mean* $\exists S',\mu'$ *such that* $(S,\mu)\to(S',\mu')$*, and* $(S,\mu)\not\to$ *for the negation of* $(S,\mu)\to$*.*

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 S and by T, that is $\mu'\backslash\mu\cup\mu''\backslash\mu\cup(\mu'\cap\mu'')$. Then for instance the scheduled programs may be written $P_i'=\texttt{when}\ s_i\ \texttt{do}\ P_i$ where s_i is the "*proceed*" signal for program P_i, set up by the scheduler. The following program

$$Sched\ {}^{t}_{n} = i:=0\,;\ \texttt{while}\ tt\ \texttt{do}\ i\ :=\ [i+1]_{\mathrm{mod}\ n}; k\ :=\ 0;\\ \texttt{while}\ k<t\ \texttt{do}\ s_i:=tt; s_i:=ff; k:=k+1$$

describes a scheduler for a system of n threads, implementing round robin with time slice t, provided that all the s_i's are initially false. It is easy to imagine how to program other scheduling policies in a similar style.

The typing rules for the new operators are given in Figure 5. The side-conditions in rules (Control) and (When) 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). On the other hand, the condition that the guard of the `when` statement should affect its guard type may seem superfluous at first sight, since a `when` statement can never be followed (in sequential composition) by any other system. The reason for this condition is that in a controlled system $S[T]$, a *blocked* behaviour of the controller S can create interferences if the controlled system T is low, and this blocked behaviour of S may be due to a `when` statement. Consider for instance the system $S[T]$ where P is a high program that does at least one step:

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

We let the reader check that this system can lead to interference. Now if the `when` statement S were allowed to have type $(L, L)\, cmd$, the whole system $S[T]$ would be typable.

As regards the rule (Control), the condition $\tau \geq \sigma$ excludes for instance – if the security levels are L and H – systems $S[T]$ whose unique type is $(L, H)\, cmd$. Consider for instance the controlled system $S'[T']$, where:

$$S' = \texttt{while}\ x_H = 0\ \texttt{do}\ \texttt{nil}$$
$$T' = y_L := 0\ ;\ y_L := y_L + 1$$

Here again there is a possible interference due to the blocking of the controller after one step if the guard of the loop is false. Note that the only possible type of $S'[T']$ would be indeed $(L, H)\, cmd$, since it affects a low variable and has a high loop guard.

To extend our noninterference result to the new setting, we also need to restrict the typing rule for conditional branching, recording the tested expression as a guard (note the similarity with the rule for the `when` statement):

(Cond-Strict)

$$\frac{\Gamma \vdash e : \theta, \quad \Gamma \vdash P : (\tau, \sigma)\, cmd, \quad \Gamma \vdash Q : (\tau, \sigma)\, cmd, \quad \theta \leq \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q : (\tau, \theta \sqcup \sigma)\, cmd}$$

This rules out for instance the thread α of our initial example (4), because a low assignment can no longer be performed after a high test.

It is easy to check that the Subject Reduction Theorem and the Confinement Lemma extend to the new language. Similarly, the definitions of (Γ, ω)-bisimulation and ω-boundedness remain formally the same as those for the base language (modulo the replacement of programs by systems). Obviously, the Guard Safety Lemma may now be strengthened into:

Lemma 4.2 (Strong Guard Safety)
If $\Gamma \vdash S : (\tau, \sigma)\, cmd$ then every loop, conditional or when statement guard in S has type θ in Γ, with $\theta \leq \sigma$.

Lemma 4.3 (Deterministic Behaviour of ω-Guarded Systems)
If S is ω-guarded in Γ and $\mu =^{\omega}_{\Gamma} \nu$, then $(S,\mu) \to (S',\mu')$ implies $(S,\nu) \to (S',\nu')$, with $\mu' =^{\omega}_{\Gamma} \nu'$.

We are now able to generalise our noninterference result.

Theorem 4.4. (Extended noninterference)
If S is typable in Γ, then $(S,\mu) \approx_{\Gamma} (S,\nu)$ for any μ,ν such that $\mu =_{\Gamma} \nu$.

Proof: We define inductively the relation $\mathcal{R}_1^{\Gamma,\omega}$ as follows: $(S,\mu)\,\mathcal{R}_1^{\Gamma,\omega}\,(T,\nu)$ if and only if S and T are typable, $\mu =^{\omega}_{\Gamma} \nu$ and one of the following holds:

1. $(S,\mu)\,\mathcal{R}_0^{\Gamma,\omega}\,(T,\nu)$, where $\mathcal{R}_0^{\Gamma,\omega}$ is the relation considered in the proof of Theorem 3.13
2. $(S,\mu) \approx^{\omega}_{\Gamma} (T,\nu)$
3. $S = T$ and S is ω-bounded
4. $S = S_0 \parallel S_1\,,\ T = T_0 \parallel T_1$ and $(S_i,\mu)\mathcal{R}_1^{\Gamma,\omega}\,(T_i,\nu)$
5. $S = \texttt{when}\ e\ \texttt{do}\ S_1$, $T = \texttt{when}\ e\ \texttt{do}\ T_1$, $\Gamma(e) \leq \omega$ and $(S_1,\mu)\mathcal{R}_1^{\Gamma,\omega}\,(T_1,\nu)$

Then we show that $\mathcal{R}_1^{\Gamma,\omega}$ is a (Γ,ω)-bisimulation. In Clause 3, for the case of the control construct, we use the Lemma 4.3. □

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 [3], 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 [8] that *covert channels*, that is implicit information flows of the kind considered here, 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 here to review the various approaches. We focussed on the approach of Volpano et al., as it applies to a fairly standard language, which can be assumed to be the kernel of more sophisticated practical languages.

The question of secure flow and noninterference has also started to be investigated in the setting of process calculi, and in particular in mobile process calculi [6], [7], [10] and [5]. The treatment in the first two papers seems however 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 [7], 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 last two papers, [10] and [5], 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.

As concerns noninterference in the presence of scheduling policies, the most popular approach has been so far the probabilistic one, taken for instance in [14] and [11]. Our stand here was to handle scheduling within a possibilistic setting.

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 [13]. Current work is also oriented towards the treatment of more realistic languages, as advocated for instance in [9], including exceptions and some form of higher-order.

Acknowledgements. We would like to thank the anonymous referees for helpful comments.

References

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

[2] Gérard Boudol and Ilaria Castellani. Noninterference for concurrent programs. Research report, INRIA, 2001.

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

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

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

[6] 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.

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

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

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

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

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

[12] 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.

[13] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT'97*, number 1214 in LNCS, pages 607–621, 1997.
[14] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3), 1999.
[15] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.