

Typing Noninterference for Reactive Programs

Ana Almeida Matos, Gérard Boudol and Ilaria Castellani

N° 5594

June 2005

Thème COM



*Rapport
de recherche*

Typing Noninterference for Reactive Programs

Ana Almeida Matos, Gérard Boudol and Ilaria Castellani

Thème COM — Systèmes communicants
Projet Mimosa

Rapport de recherche n° 5594 — June 2005 — 38 pages

Abstract: We propose a type system to enforce the security property of *noninterference* in a core reactive language, obtained by extending the imperative language of Volpano, Smith and Irvine with reactive primitives manipulating broadcast signals and with a form of “scheduled” parallelism. Due to the particular nature of reactive computations, the definition of noninterference has to be adapted. We give a formulation of noninterference based on bisimulation. Our type system is inspired by that introduced by Boudol and Castellani, and independently by Smith, to cope with timing leaks in a language for parallel programs with scheduling. We establish the soundness of this type system with respect to our notion of noninterference.

Key-words: Reactive programming, noninterference, bisimulation, types.

Typage de la non-interférence pour les programmes réactifs

Résumé : Nous proposons un système de types garantissant la propriété de *non-interférence* pour un langage réactif simple, obtenu par l'ajout au langage impératif de Volpano, Smith et Irvine de primitives réactives pour la diffusion et la manipulation de signaux et d'une forme de parallélisme ordonné. A cause de la nature particulière des calculs réactifs, la définition de noninterférence doit être adaptée. Nous en donnons une formulation basée sur la notion de bisimulation. Notre système de types s'inspire de celui introduit par Boudol et Castellani, et indépendamment par Smith, pour traiter des "fuites temporelles" (timing leaks) dans un langage pour les programmes parallèles ordonnés. Nous établissons la correction du système de types par rapport à la notion de non-interférence.

Mots-clés : Programmation réactive, non-interférence, bisimulation, types.

1 Introduction

To be widely accepted and deployed, the mobile code technology has to provide formal guarantees regarding the various security issues that it raises. For instance, foreign code should not be allowed to corrupt, or even simply to get knowledge of secret data owned by its execution context. Similarly, a supposedly trusted code should be checked for not disclosing private data to public knowledge. In [5] we have introduced a core programming model for mobile code called ULM, advocating the use of a *locally synchronous* programming style [3, 11] in a *globally asynchronous* computing context. It is therefore natural to examine the security issues from the point of view of this programming model. In this paper, we address some of these issues, and more specifically the non-disclosure property, for a simplified version of the ULM language. We recall the main features of the synchronous programming style, in its control-oriented incarnation:

Broadcast signals Program components react according to the presence or absence of signals, by computing and emitting signals that are broadcast to all components of a given “synchronous area”.

Suspension Program components may be in a suspended state, because they are waiting for a signal which is absent at the moment where they get the control.

Preemption There are means to abort the execution of a program component, depending on the presence or absence of a signal.

Instants Instants are successive periods of the execution of a program, where the signals are consistently seen as present or absent by all components.

The so-called *reactive* variant of the synchronous programming style, designed by Boussinot, has been implemented in a number of languages and used for various applications, see [9, 8]. This differs from the synchronous language ESTEREL [4], for instance in the way absence of signals is dealt with: in reactive programming, the absence of a signal can only be determined at the end of the current instant, and reaction is postponed to the next instant. In this way, the causal paradoxes that arise in some ESTEREL programs can be avoided, making reactive programming well suited for systems where concurrent components may be dynamically added or removed, as it is the case with mobile code.

We consider here a core reactive language, which is a subset of ULM that extends the sequential language of [19] with reactive primitives and with an operator of alternating parallel composition (incorporating a fixed form of scheduling). As expected, these new constructs add expressive power to the language and induce new forms of security leaks. Moreover, the two-level nature of reactive computations, which evolve both within instants and across instants, introduces new subtleties in the definition of noninterference. We give a formulation of noninterference based on bisimulation, as is now standard [16, 15, 17, 6]. We define a type system to enforce this property of noninterference, along the lines of that proposed by Boudol and Castellani [7], and independently by Smith [17], for a language for parallel programs with scheduling. In this approach, types impose constraints on the relation between the security levels of tested variables and signals, on one side, and those of written variables and emitted signals, on the other side.

Let us briefly recall the intuition about noninterference. The idea is that in a system with multiple security levels, information should only be allowed to flow from lower to higher levels [10]. As usual, we assume security levels to form a lattice. However, in most of our examples we shall use only two security levels, *low* (public, L) and *high* (secret, H). Security levels are attributed to variables and signals, and we will use subscripts to specify them (eg. x_H is a variable of high level). In a sequential imperative language, an insecure flow of information, or interference, occurs when the initial values of high variables influence the final values of low variables. The simplest case of insecure flow is the assignment of the value of a high variable to a low variable, as in $y_L := x_H$. This is called *explicit (insecure) flow*. More subtle kinds of flow, called *implicit flows*, may be induced by the flow of control. A typical example is the program (if $x_H = 0$ then $y_L := 0$ else nil), where the value of y_L may give information about x_H .

Other programs may be considered as secure or not depending on the context in which they appear. For instance, the program

$$(\text{while } x_H \neq 0 \text{ do nil}); y_L := 0 \quad (1)$$

is safe in a sequential setting (since whenever it terminates it produces the same value for y_L), whereas it becomes critical in the presence of asynchronous parallelism (as explained e.g. in [6, 7]).

When moving to a reactive setting we must reconsider the security of programs with respect to the reactive contexts. In the ULM model there are two kinds of parallel composition:

1. The global asynchronous composition of “reactive machines”: this is similar to the parallel composition usually considered in the literature (see for instance [16, 7, 17]), with the difference that no specific scheduling is assumed at this level. We do not consider this global composition here, and we expect it could be dealt with in a standard compositional manner.
2. The local synchronous composition of threads, within each reactive machine (locally synchronous area). Like in the implementation of reactive programming [8], we assume a deterministic cooperative scheduling discipline on threads. It is well known that scheduling introduces new possibilities of flow (see e.g. [16, 15]), and this will indeed be the case with the scheduling that we adopt here. In fact, we shall see that loops with high conditions, followed by low assignments (like program (1) above) can be dangerous also in a reactive setting.

Another issue we are faced with when addressing the security of reactive programs is their ability to suspend while waiting for an absent signal, thus giving rise to a special event called *instant change*. One of the effects of an instant change is to reset all signals to “absent”. With the constructs of the language we are able to write (for any security level) a program `pause`, whose behaviour is to suspend for the current instant, and terminate at the beginning of the next instant (see Section 2.2). Then we may write the following program:

$$\text{emit } a_L; \text{ if } x_H = 0 \text{ then nil else pause} \quad (2)$$

This program starts by emitting signal a_L . Then, depending on the value of x_H , it may either terminate within an instant, in which case a_L remains present, or suspend and change instant, in which case a_L is erased. However, since instant changes are not statically predictable, we do not

consider as observable the withdrawal of low signals that occurs at instant change. Consequently, we consider (2) as safe. More complex examples of reactive programs will be given in Section 3, once the language has been introduced. The new phenomena occurring in reactive programming will lead us to modify the usual definition of noninterference, as well as the type system used to ensure it.

The rest of the paper is organized as follows. In Section 2 we introduce the language and its operational semantics. Section 3 presents the type system and some properties of typed programs, including subject reduction. We then define noninterference by means of a bisimulation relation and prove the soundness of our type system. This paper is the full version of [2], completed with proofs.

2 The language

2.1 Syntax

We consider two infinite and disjoint sets of *variables* and *signals*, Var and Sig , ranged over by x, y, z and a, b, c respectively. We then let $Names$ be the union $Var \cup Sig$, ranged over by n, m . The set Exp of expressions includes booleans and naturals with the usual operations, but no signals. For convenience we have chosen to present the type system only in Section 3.1. However types, or more precisely *security levels*, ranged over by δ, θ, σ , already appear in the syntax of the language. Security levels constitute what we call *simple types*, and are used to type expressions and declared signals. In Section 3 we will see how more complex types for variables, signals and programs may be built from simple types.

The language of *processes* $P, Q \in Proc$ is defined by:

$$P ::= \text{nil} \mid x := e \mid \text{let } x : \delta = e \text{ in } P \mid \text{if } e \text{ then } P \text{ else } Q \mid \text{while } e \text{ do } P \mid P ; Q \\ \text{emit } a \mid \text{local } a : \delta \text{ in } P \mid \text{do } P \text{ watching } a \mid \text{when } a \text{ do } P \mid (P \uparrow Q)$$

Note the use of brackets to make explicit the precedence of \uparrow (which, as we shall see, is a non associative operator). The construct $\text{let } x : \delta = e \text{ in } Q$ binds free occurrences of variable x in Q , whereas $\text{local } a : \delta \text{ in } Q$ binds free occurrences of signal a in Q . The free variables and signals of a program P , noted $\text{fv}(P)$ and $\text{fs}(P)$ respectively, are defined in the usual way.

2.2 Operational Semantics

The operational semantics is defined on *configurations*, which are quadruples $C = \langle \Gamma, S, E, P \rangle$ composed of a type-environment Γ , a variable-store S , a signal-environment E and a program P . The type-environment is a mapping from names to the appropriate types. Types for variables and signals, formally introduced in Section 3.1, have the form $\delta \text{ var}$ and $\delta \text{ sig}$ respectively. We denote the update of Γ by $\{x : \delta \text{ var}\}\Gamma$ or $\{a : \delta \text{ sig}\}\Gamma$. A variable-store is a mapping from variables to values. By abuse of language we denote by $S(e)$ the atomic evaluation of the expression e under S , which we assume to always terminate and to produce no side effects. We denote by $\{x \mapsto S(e)\}S$ the update or extension of S with the value of e for the variable x , depending on whether the variable is present or not in the domain of S . The signal-environment is the set of signals which are considered to be present. We restrict our attention to *well-formed configurations* $C = \langle \Gamma, S, E, P \rangle$, satisfying

$$\begin{array}{c}
\text{(WHEN-SUS}_1\text{)} \frac{a \notin E}{(E, \text{when } a \text{ do } P)\ddagger} \quad \text{(WHEN-SUS}_2\text{)} \frac{(E, P)\ddagger}{(E, \text{when } a \text{ do } P)\ddagger} \\
\text{(WATCH-SUS)} \frac{(E, P)\ddagger}{(E, \text{do } P \text{ watching } a)\ddagger} \\
\text{(SEQ-SUS)} \frac{(E, P)\ddagger}{(E, P; Q)\ddagger} \quad \text{(PAR-SUS)} \frac{(E, P)\ddagger \quad (E, Q)\ddagger}{(E, P \uparrow Q)\ddagger}
\end{array}$$

Figure 1: Suspension predicate

the conditions $\text{fv}(P) \subseteq \text{dom}(\Gamma)$, $\text{fv}(P) \subseteq \text{dom}(S)$ and $\text{dom}(S) \cup E \subseteq \text{dom}(\Gamma)$. We shall see in Section 2.2.3 that well-formedness is preserved by execution.

A distinguishing feature of reactive programs is their ability to *suspend* while waiting for a signal. The suspension predicate is defined inductively on pairs (E, P) by the rules in Figure 1. Suspension is introduced by the construct `when a do P`, in case signal a is absent. The suspension of a program P is propagated to certain contexts, namely $P; Q$, `do P watching a`, `when a do P` and $P \uparrow Q$, which we call *suspendable processes*. We extend suspension to configurations by letting $\langle \Gamma, S, E, P \rangle \ddagger$ if $\langle E, P \rangle \ddagger$.

There are two forms of transitions between configurations: simple *moves*, denoted by the arrow $C \rightarrow C'$, and *instant changes*, denoted by $C \leftrightarrow C'$. These are collectively referred to as *steps* and denoted by $C \mapsto C'$. The reflexive and transitive closures of these transition relations are denoted with a ‘*’ as usual. An *instant* is a sequence of moves leading to termination or suspension.

2.2.1 Moves

The operational rules for imperative and reactive constructs are given in Figures 2 and 3 respectively. The functions $\text{newv}(N)$ and $\text{news}(N)$ take a finite set of names N and return a fresh name not in N , respectively a variable and a signal. They are used to preserve determinism in the language. The notation $\{n/m\}P$ stands for the (capture avoiding) substitution of m by n in P .

The rules for the imperative constructs are standard. Termination is dealt with by reduction to `nil`. Some comments on the rules for the reactive constructs are in order. Signal emission adds a signal to the signal-environment. The local signal declaration is standard. The `watching` construct allows the execution of its body until an instant change occurs; the execution will then resume or not at the next instant depending on the presence of the signal (as explained in more detail below). As for the `when` construct, it executes its body if the tested signal is present, and suspends otherwise.

Alternating parallel composition \uparrow implements a co-routine mechanism. It executes its left component until termination or suspension, and then gives control to its right component, provided this one is not already suspended. Note that \uparrow incorporates a *cooperative scheduling* discipline. To workn fairly, it requires each thread to “play the game” and yield the control after a finite number of steps : in $P \uparrow Q$, if P does not terminate then Q will never get the control.

$$\begin{array}{l}
\text{(ASSIGN-OP)} \quad \langle \Gamma, S, E, x := e \rangle \rightarrow \langle \Gamma, \{x \mapsto S(e)\} S, E, \text{nil} \rangle \\
\\
\text{(SEQ-OP}_1\text{)} \quad \langle \Gamma, S, E, \text{nil}; Q \rangle \rightarrow \langle \Gamma, S, E, Q \rangle \\
\\
\text{(SEQ-OP}_2\text{)} \quad \frac{\langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, P; Q \rangle \rightarrow \langle \Gamma', S', E', P'; Q \rangle} \\
\\
\text{(LET-OP)} \quad \frac{x' \notin \text{dom}(\Gamma)}{\langle \Gamma, S, E, \text{let } x : \delta = e \text{ in } P \rangle \rightarrow \langle \{x' : \delta \text{ var}\} \Gamma, \{x' \mapsto S(e)\} S, E, \{x'/x\} P \rangle} \\
\\
\text{(COND-OP}_1\text{)} \quad \frac{S(e) = \text{true}}{\langle \Gamma, S, E, \text{if } e \text{ then } P \text{ else } Q \rangle \rightarrow \langle \Gamma, S, E, P \rangle} \\
\\
\text{(COND-OP}_2\text{)} \quad \frac{S(e) = \text{false}}{\langle \Gamma, S, E, \text{if } e \text{ then } P \text{ else } Q \rangle \rightarrow \langle \Gamma, S, E, Q \rangle} \\
\\
\text{(WHILE-OP}_1\text{)} \quad \frac{S(e) = \text{true}}{\langle \Gamma, S, E, \text{while } e \text{ do } P \rangle \rightarrow \langle \Gamma, S, E, P; \text{while } e \text{ do } P \rangle} \\
\\
\text{(WHILE-OP}_2\text{)} \quad \frac{S(e) = \text{false}}{\langle \Gamma, S, E, \text{while } e \text{ do } P \rangle \rightarrow \langle \Gamma, S, E, \text{nil} \rangle}
\end{array}$$

Figure 2: Operational semantics of imperative constructs

$$\begin{array}{l}
\text{(EMIT-OP)} \quad \langle \Gamma, S, E, \text{emit } a \rangle \rightarrow \langle \Gamma, S, \{a\} \cup E, \text{nil} \rangle \\
\text{(LOCAL-OP)} \quad \frac{a' \notin \text{dom}(\Gamma)}{\langle \Gamma, S, E, \text{local } a : \delta \text{ in } P \rangle \rightarrow \langle \{a' : \delta \text{ sig}\} \Gamma, S, E, \{a'/a\} P \rangle} \\
\text{(WATCH-OP}_1\text{)} \quad \langle \Gamma, S, E, \text{do nil watching } a \rangle \rightarrow \langle \Gamma, S, E, \text{nil} \rangle \\
\text{(WATCH-OP}_2\text{)} \quad \frac{\langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, \text{do } P \text{ watching } a \rangle \rightarrow \langle \Gamma', S', E', \text{do } P' \text{ watching } a \rangle} \\
\text{(WHEN-OP}_1\text{)} \quad \frac{a \in E}{\langle \Gamma, S, E, \text{when } a \text{ do nil} \rangle \rightarrow \langle \Gamma, S, E, \text{nil} \rangle} \\
\text{(WHEN-OP}_2\text{)} \quad \frac{a \in E \quad \langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, \text{when } a \text{ do } P \rangle \rightarrow \langle \Gamma', S', E', \text{when } a \text{ do } P' \rangle} \\
\text{(PAR-OP}_1\text{)} \quad \langle \Gamma, S, E, \text{nil } \dot{\vee} Q \rangle \rightarrow \langle \Gamma, S, E, Q \rangle \\
\text{(PAR-OP}_2\text{)} \quad \frac{\langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle}{\langle \Gamma, S, E, P \dot{\vee} Q \rangle \rightarrow \langle \Gamma', S', E', P' \dot{\vee} Q \rangle} \\
\text{(PAR-OP}_3\text{)} \quad \frac{\langle E, P \rangle \ddagger \quad \neg \langle E, Q \rangle \ddagger}{\langle \Gamma, S, E, P \dot{\vee} Q \rangle \rightarrow \langle \Gamma, S, E, Q \dot{\vee} P \rangle}
\end{array}$$

Figure 3: Operational semantics of reactive constructs

Example 1 (Alternating parallel composition) *In this example three threads are queuing for execution in an empty signal-environment (left column). Here and in the following underbraces indicate suspension of the executing thread. The emission of signal a by the third process unblocks the first of the suspended processes, enabling them to execute one after the other and then reach termination.*

\rightarrow	$\{\}$	$((\text{when } a \text{ do emit } b) \frown (\text{when } b \text{ do emit } c)) \frown \text{emit } a$
\rightarrow^*	$\{\}$	$\text{emit } a \frown ((\text{when } a \text{ do emit } b) \frown (\text{when } b \text{ do emit } c))$
\rightarrow^*	$\{a\}$	$(\text{when } a \text{ do emit } b) \frown (\text{when } b \text{ do emit } c)$
\rightarrow^*	$\{a, b\}$	$\text{when } b \text{ do emit } c$
\rightarrow^*	$\{a, b, c\}$	nil

The operator \frown is clearly non commutative with respect to execution traces. It is also non associative, as can be seen by looking at the pair of programs

$$((\text{when } a \text{ do emit } b) \frown \text{emit } a) \frown \text{emit } c \quad \text{and} \quad (\text{when } a \text{ do emit } b) \frown (\text{emit } a \frown \text{emit } c)$$

Starting from an empty signal environment, the first program will emit the signals in the order a, b, c , while the second will emit them in the order a, c, b .

We have seen that suspension of a thread may be lifted during an instant upon emission of the signal by another thread in the pool. This is no longer possible in a program in which all threads are suspended. When this situation is reached, an instant change occurs.

2.2.2 Instant changes

Suspension of a configuration marks the end of an instant. At this point, all suspended subprocesses of the form $\text{do } P \text{ watching } a$ whose tested signal a is present are killed, and all signals are reset to absent (that is, the new signal-environment is the empty set). Indeed, the `watching` construct provides a mechanism to recover from suspension and from deadlock situations originated by `when` commands (as will be illustrated by the causality cycle example below). The semantics of *instant changes* is defined in Figure 4. The function $\lfloor P \rfloor_E$ is meant to be applied to suspended processes (see Figure 1) and therefore is defined only for them.

Instant changes are programmable; we hinted in the Introduction at the possibility of encoding a primitive that enforces suspension of a thread until instant change. This primitive, which we call `pause`, is defined as follows.

Example 2 (pause) *Here the local declaration of signal a ensures that the signal cannot be emitted outside the scope of its declaration, and therefore that the program will suspend when reaching the subprogram `when a do nil`. At this point, the presence of b (replaced upon creation by a fresh signal b') is checked, and since it has been emitted, the subprogram `when a do nil` (where a has been replaced by a fresh signal a') is aborted at the beginning of the next instant.*

\rightarrow^*	$\{\}$	$\text{local } a : \delta \text{ in } (\text{local } b : \theta \text{ in } (\text{emit } b; \text{do } (\text{when } a \text{ do nil}) \text{ watching } b))$
\rightarrow^*	$\{\}$	$\text{emit } b'; \text{do } (\text{when } a' \text{ do nil}) \text{ watching } b'$
\rightarrow^*	$\{b'\}$	$\text{do } (\text{when } a' \text{ do nil}) \text{ watching } b'$
\hookrightarrow	$\{\}$	nil

In the following examples we shall assume programs to run in the empty signal environment if not otherwise specified.

Note that the program `pause` does not suspend immediately but only after performing a few “administrative moves”. This implies for instance that the program

$$(\text{pause}; P) \uparrow (\text{pause}; Q)$$

evolves to the program $Q \uparrow P$ after a change of instant, since the second component gets the control before the suspension. On the other hand, no switch of control occurs in the program

$$(\text{pause}; P) \uparrow (\text{when } a \text{ do } Q)$$

since in this case the second component suspends immediately. Hence this program will evolve to $P \uparrow (\text{when } a \text{ do } Q)$ at instant change.

The following is an example where an instant change breaks a causality cycle:

$$\text{emit } a; ((\text{when } b \text{ do emit } c) \uparrow (\text{do } (\text{when } c \text{ do emit } b) \text{ watching } a); \text{emit } b)$$

Here the whole program suspends after the emission of a . Then, since a is present, the body of the `watching` construct is killed and a new instant starts, during which b is emitted, thus unblocking the other thread and allowing c to be emitted. This is an example of a deadlock situation which is exited at the end of instant thanks to the `watching` construct.

2.2.3 Execution paths

A configuration $C = \langle \Gamma, S, E, P \rangle$ is *alive* if it can perform a step, what we denote by $C \mapsto$. Otherwise it is *terminated*, what we denote by $C \not\mapsto$. If $C = \langle \Gamma, S, E, P \rangle$ is able to perform a step, the form of this step depends on whether P is suspended or not in the environment E . A computation has the general form:

$$\langle \Gamma, S, E, P \rangle \rightarrow^* \langle \Gamma_1, S_1, E_1, P_1 \rangle \hookrightarrow \langle \Gamma_1, S_1, \emptyset, [P_1]_{E_1} \rangle \rightarrow^* \langle \Gamma_2, S_2, E_2, P_2 \rangle \hookrightarrow \dots$$

We establish now a few properties of computations. Given a configuration $\langle \Gamma, S, E, P \rangle$, we use the term *memory* to refer to the pair (S, E) . It is easy to see from the semantic rules that computations may affect a memory only by updating it or by extending it with a fresh variable. Similarly, they may affect a type-environment only by extending it with a fresh name. These facts are summed up in the following proposition, which we state without proof.

Proposition 2.1 (Simple properties of computations)

1. If $\langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle$, then $\Gamma' = \Gamma$ or $\Gamma' = \{n : \delta \text{ name}\} \Gamma$, where $n \notin \text{dom}(\Gamma)$.
2. If $\langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle$, then $\text{dom}(S') = \text{dom}(S)$ or $\text{dom}(S') = \text{dom}(S) \cup \{x\}$, for some $x \notin \text{dom}(S)$, and $E' = E$ or $E' = E \cup \{a\}$ for some $a \in \text{dom}(\Gamma) \setminus E$.

We show now that computations preserve well-formedness of configurations. We recall that a configuration $C = \langle \Gamma, S, E, P \rangle$ is well-formed if $\text{fs}(P) \subseteq \text{dom}(\Gamma)$, $\text{fv}(P) \subseteq \text{dom}(S)$ and $\text{dom}(S) \cup E \subseteq \text{dom}(\Gamma)$.

Proposition 2.2 (Well-formedness is preserved along execution)

If C is a well-formed configuration and $C \mapsto C'$ then C' is also a well-formed configuration.

Proof Let $C = \langle \Gamma, S, E, P \rangle$ and $C' = \langle \Gamma', S', E', P' \rangle$. We must show that C' satisfies the properties $\text{fs}(P') \subseteq \text{dom}(\Gamma')$, $\text{fv}(P') \subseteq \text{dom}(S')$ and $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$. We distinguish the two cases $C \hookrightarrow C'$ and $C \rightarrow C'$.

1. *Instant change.* If $C \hookrightarrow C'$ then $\Gamma' = \Gamma, S' = S, E' = \emptyset$ and $P' = \lfloor P \rfloor_E$. It is easy to see that $\text{fv}(\lfloor P \rfloor_E) \subseteq \text{fv}(P)$. Then the properties for C' follow immediately from those for C .
2. *Simple move.* Suppose now $C \rightarrow C'$. We prove the required properties by induction on the proof of the transition. There are several base cases to consider. Indeed, the only cases where induction is used are those of rules (SEQ-OP₂), (WATCH-OP₂), (WHEN-OP₂) and (PAR-OP₂). Note that in all base cases apart from (LET-OP), we have $\text{fv}(P') \subseteq \text{fv}(P)$ and $\text{dom}(S') = \text{dom}(S)$, so the property $\text{fv}(P') \subseteq \text{dom}(S')$ is trivial. Similarly, in all base cases apart from (LET-OP) and (LOCAL-OP), we have $\text{fs}(P') \subseteq \text{fs}(P)$ and $\text{dom}(\Gamma') = \text{dom}(\Gamma)$, so the property $\text{fs}(P') \subseteq \text{dom}(\Gamma')$ is trivial. As for the property $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$, it is trivial in all cases where $\Gamma' = \Gamma, S' = S$ and $E' = E$. We examine the remaining cases.

- (ASSIGN-OP) Here $\Gamma' = \Gamma, \text{dom}(S') = \text{dom}(S)$ and $E' = E$, hence the property $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$ follows immediately from that for C .
- (SEQ-OP₂) Here $P = P_1; P_2, P' = P'_1; P_2$, and the transition $C \rightarrow C'$ is deduced from $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. By induction $\text{fs}(P'_1) \subseteq \text{dom}(\Gamma'), \text{fv}(P'_1) \subseteq \text{dom}(S')$ and $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$. Hence the last property for C' is already given. Now, since C is well-formed, we know that $\text{fs}(P_2) \subseteq \text{dom}(\Gamma)$ and $\text{fv}(P_2) \subseteq \text{dom}(S)$. By Proposition 2.1 $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$ and $\text{dom}(S) \subseteq \text{dom}(S')$, whence $\text{fs}(P'_1; P_2) = \text{fs}(P'_1) \cup \text{fs}(P_2) \subseteq \text{dom}(\Gamma')$ and $\text{fv}(P'_1; P_2) = \text{fv}(P'_1) \cup \text{fv}(P_2) \subseteq \text{dom}(S')$.
- (LET-OP) Here $P = \text{let } x : \delta = e \text{ in } P_1$ and $P' = \{x'/x\}P_1$, for some x' not in $\text{dom}(\Gamma)$ and thus not in $\text{dom}(S)$ nor in $\text{fv}(P)$. We have $\Gamma' = \{x' : \delta \text{ var}\}\Gamma, S' = \{x' \mapsto S(e)\}S, E' = E$. Since $\text{fs}(P') = \text{fs}(P)$ and $\text{fv}(P') = \text{fv}(P) \cup \{x'\}$, it follows that $\text{fs}(P') \subseteq \text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$ and $\text{fv}(P') \subseteq \text{dom}(S) \cup \{x'\} = \text{dom}(S')$. Similarly, we have $\text{dom}(S') \cup E' = \text{dom}(S) \cup \{x'\} \cup E \subseteq \text{dom}(\Gamma) \cup \{x'\} = \text{dom}(\Gamma')$.
- (EMIT-OP) Here $P = \text{emit } a$ and we have $\Gamma' = \Gamma, S' = S$ and $E' = E \cup \{a\}$. Since C is well-formed, we know that $\text{fs}(P) \subseteq \text{dom}(\Gamma)$, hence $a \in \text{dom}(\Gamma)$. We can then conclude that $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$.
- (LOCAL-OP) Here $P = \text{local } a : \delta \text{ in } P_1$ and $P' = \{a'/a\}P_1$ for some a' not in $\text{dom}(\Gamma)$. Since $\Gamma' = \{a' : \delta \text{ sig}\}\Gamma, S' = S$ and $E' = E$, we have $\text{fs}(P') = \text{fs}(P) \cup \{a'\} \subseteq \text{dom}(\Gamma) \cup \{a'\} = \text{dom}(\Gamma')$ and $\text{dom}(S') \cup E' = \text{dom}(S) \cup E \subseteq \text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$.

- (WATCH-OP₂) Here $P = \text{do } P_1 \text{ watching } a$, $P' = \text{do } P'_1 \text{ watching } a$ and the transition $C \rightarrow C'$ is deduced from $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. By induction we have $\text{fs}(P'_1) \subseteq \text{dom}(\Gamma')$, $\text{fv}(P'_1) \subseteq \text{dom}(S')$ and $\text{dom}(S') \cup E' \subseteq \text{dom}(\Gamma')$. Thus the third property is given. Moreover $\text{fv}(P') = \text{fv}(P'_1)$, so we only have to prove the first property. By Proposition 2.1 $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$. Since C is well-formed, we know that $\text{fs}(P) \subseteq \text{dom}(\Gamma)$, thus $a \in \text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$. Whence $\text{fs}(P') = \text{fs}(P'_1) \cup \{a\} \subseteq \text{dom}(\Gamma') \cup \{a\} = \text{dom}(\Gamma')$.
- The cases of rules (WHEN-OP₂) and (PAR-OP₂) are similar.

□

By virtue of this result, we may always assume configurations to be well-formed. We shall generally do so without explicitly mentioning it. As a first consequence of well-formedness, we will show that a configuration is terminated if and only if the executing process is syntactically equal to nil . To prove this fact we shall also need the following assumption about expression evaluation, already mentioned in Section 2.2.

Assumption 2.3 *For any S , e such that $\text{fv}(e) \subseteq \text{dom}(S)$, the value $S(e)$ is defined.*

Proposition 2.4 *A configuration $C = \langle \Gamma, S, E, P \rangle$ is terminated if and only if $P = \text{nil}$.*

Proof We prove that $P \neq \text{nil}$ implies $C \not\rightarrow$, by induction on the structure of P . Recall that by definition $C \hookrightarrow$ if and only if $\langle E, P \rangle \ddagger$. We examine some sample cases.

- $P = x := e$. Since C is well-formed, we have $\text{fv}(e) \in \text{dom}(S)$ and thus by Assumption 2.3 the value $S(e)$ is defined. Then $C \rightarrow \langle \Gamma, \{x \mapsto S(e)\}S, E, \text{nil} \rangle$ by (ASSIGN-OP).
- $P = P_1 ; P_2$. If $P_1 = \text{nil}$, then $C \rightarrow \langle \Gamma, S, E, P_2 \rangle$ by (SEQ-OP₁). If $P_1 \neq \text{nil}$ then by induction either $\langle E, P_1 \rangle \ddagger$ or $\exists \Gamma', S', E', P'_1$ such that $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. In the first case we have $\langle E, P_1 ; P_2 \rangle \ddagger$ by (SEQ-SUS) and $C \hookrightarrow \langle \Gamma', S', E', \lfloor P_1 \rfloor_E ; P_2 \rangle$ by (INSTANT-OP), while in the latter we have $C \rightarrow \langle \Gamma', S', E', P'_1 ; P_2 \rangle$ by (SEQ-OP₂).
- $P = \text{let } x : \delta = e \text{ in } P_1$. Since $\text{newv}(N)$ is a total function, we have $C \rightarrow$ by (LET-OP).
- $P = \text{if } e \text{ then } P_1 \text{ else } P_2$. By well-formedness and Assumption 2.3 we know that $S(e)$ is defined (and we may assume it to be a boolean value, by some implicit typing). Then we deduce $C \rightarrow$, using rule (COND-OP₁) or rule (COND-OP₂) depending on the value $S(e)$.
- $P = \text{while } e \text{ do } P_1$. Similar to $P = \text{if } e \text{ then } P_1 \text{ else } P_2$.
- $P = \text{do } P_1 \text{ watching } a$. Similar to $P = P_1 ; P_2$.
- $P = \text{when } a \text{ do } P_1$. If $a \notin E$, then we have $\langle E, \text{when } a \text{ do } P_1 \rangle \ddagger$ by (WHEN-SUS) and $C \hookrightarrow \langle \Gamma', S', E', \lfloor \text{when } a \text{ do } P_1 \rfloor_E \rangle$ by (INSTANT-OP). Assume now $a \in E$. If $P_1 = \text{nil}$, then $C \rightarrow \langle \Gamma, S, E, \text{nil} \rangle$ by rule (WHEN-OP₁). If $P_1 \neq \text{nil}$, then we use induction exactly as in the case $P = P_1 ; P_2$.

- $P = P_1 \uparrow P_2$. If $P_1 = \text{nil}$, then $C \rightarrow \langle \Gamma, S, E, P_2 \rangle$ by (PAR-OP₁). If $P_1 \neq \text{nil}$ then by induction either $\langle E, P_1 \rangle \dagger$ or $\exists \Gamma', S', E', P'_1$ such that $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. In the latter case $C \rightarrow \langle \Gamma', S', E', P'_1 \uparrow P_2 \rangle$ by (PAR-OP₂). In the first case either also $\langle E, P_2 \rangle \dagger$ and thus $\langle E, P_1; P_2 \rangle \dagger$ by (PAR-SUS) and $C \hookrightarrow \langle \Gamma', S', E', \lfloor P_1 \rfloor_E; P_2 \rangle$ by (INSTANT-OP), or $\neg \langle E, P_2 \rangle \dagger$ and $C \rightarrow \langle \Gamma, S, E, P_2 \uparrow P_1 \rangle$ by (PAR-OP₃).

□

We are now able to prove an important property of reactive programs, namely their deterministic behaviour up to the choice of local names. If C is a configuration, let $C\{n/m\}$ be the pointwise substitution of name n for name m in all components of C .

Proposition 2.5 (Determinism) *If a configuration $C = \langle \Gamma, S, E, P \rangle$ is alive, it is either suspended, if $\langle E, P \rangle \dagger$, in which case $\exists! C' : C \hookrightarrow C'$, or active, in which case $\exists C' = \langle \Gamma', S', E', P' \rangle$ such that $C \rightarrow C'$ and for any $C'' = \langle \Gamma'', S'', E'', P'' \rangle$ such that $C \rightarrow C''$ either $C'' = C'$, or $\text{dom}(\Gamma'') \setminus \text{dom}(\Gamma) = \{n\}$, $\text{dom}(\Gamma') \setminus \text{dom}(\Gamma) = \{m\}$ and $C'' = C'\{n/m\}$.*

Proof By Proposition 2.4 we know that $P \neq \text{nil}$. If $\langle E, P \rangle \dagger$, the only rule that applies to C is (INSTANT-OP), yielding the transition $C \hookrightarrow C' = \langle \Gamma, S, \emptyset, \lfloor P \rfloor_E \rangle$. If $\neg \langle E, P \rangle \dagger$, then it may be easily checked, by inspection of the rules in Figures 2 and 3, that exactly one rule will be applicable to C , yielding a unique transition $C \rightarrow C'$ if this rule is different from (LET-OP) or (LOCAL-OP). If the rule is (LET-OP) or (LOCAL-OP), then C has an infinity of moves, one for each choice of the new name, and clearly the resulting configurations are the same up to a renaming of this name.

□

3 Noninterference

In this section we introduce our type system and prove some of its properties. We then formalise our security property as a form of self-bisimilarity, and prove that our type system guarantees this property. Finally, we compare our notion of security with a more standard one, and show that our notion is stronger in several respects (and thus closer to the notion of typability).

3.1 Type System

The role of the type system is to rule out insecure programs. Now, what should be the security notion for our language? Should it be based on a *termination sensitive* semantics (as for sequential languages [19]), where only the final values of executions are observable? Or should it rather rely on a *termination insensitive* semantics (as for the parallel languages of [16, 15, 7]), where all intermediate values of possibly nonterminating executions are taken into account? The answer is not immediately obvious, since in our language parallelism consists of a deterministic interleaving

$$\begin{array}{l}
\text{(INSTANT-OP)} \quad \frac{\langle E, P \rangle_{\ddagger}}{\langle \Gamma, S, E, P \rangle \leftrightarrow \langle \Gamma, S, \emptyset, [P]_E \rangle} \quad \text{where} \\
\begin{array}{l}
[\text{do } P \text{ watching } a]_E \stackrel{\text{def}}{=} \begin{cases} \text{nil} & \text{if } a \in E \\ \text{do } [P]_E \text{ watching } a & \text{otherwise} \end{cases} \\
[\text{when } a \text{ do } P]_E \stackrel{\text{def}}{=} \begin{cases} \text{when } a \text{ do } [P]_E & \text{if } a \in E \\ \text{when } a \text{ do } P & \text{otherwise} \end{cases} \\
[P; Q]_E \stackrel{\text{def}}{=} [P]_E; Q \\
[P \uparrow Q]_E \stackrel{\text{def}}{=} [P]_E \uparrow [Q]_E
\end{array}
\end{array}$$

Figure 4: Operational semantics of instant changes

of cooperative threads, and thus some of the problems raised by input-output semantics in a parallel setting disappear (like the non-reproducibility of results and the lack of compositionality).

However, it is well-known that in reactive programming, as well as in other kinds of parallel programming, input-output semantics is not appropriate since many useful programs (like controllers, schedulers, service providers and other programs reacting to environment requests) are explicitly designed to be persistent. On the other hand, reactive programs execute across instants, and it is generally agreed that an instantly diverging program (that is, a program which loops within an instant, also called *instantaneous loop*) should be rejected. In other words, a “good” nonterminating program should span over an infinity of instants. One could then envisage to adopt an intermediate semantics, where values are observed only at the end of instants, where computations are guaranteed to terminate if programs are well-behaved. We shall leave this question open for future investigation, and adopt here a termination insensitive semantics as in previous work on concurrent languages.

In the Introduction we illustrated the notion of (insecure) implicit flow in sequential programs. It is easy to see that similar implicit flows arise also with reactive constructs. Consider the program

$$\text{emit } c_L; (\text{do } (\text{when } a_H \text{ do emit } b_L) \text{ watching } c_L) \quad (3)$$

Whether a_H is present or not this program always terminates (in one or two instants respectively), but b_L is emitted only if a_H is present. Also the more subtle program (1) has its reactive counterpart

$$(\text{when } a_H \text{ do nil}); \text{emit } b_L \quad (4)$$

Indeed this program may be source of leaks when composed with other threads, since suspension can be lifted by the emission of signal a_H .

Consider for instance the program $\gamma \uparrow (\alpha \uparrow \beta)$ (which can be viewed as a reactive analogue of the PIN example of [16, 7]), where

$$\begin{aligned} \gamma &: \text{if PIN}_H = 0 \text{ then emit } a_H \text{ else nil} \\ \alpha &: (\text{when } a_H \text{ do nil}); \text{emit } b_L \\ \beta &: \text{emit } c_L; \text{emit } a_H \end{aligned} \quad (5)$$

If $\text{PIN}_H = 0$, then α is completely executed before β gets the control, and b_L is emitted before c_L . If $\text{PIN}_H \neq 0$, then α suspends and β executes, emitting c_L and then a_H , thus unblocking α . In this case b_L is emitted after c_L . We should point out that thread γ is not essential here (it is included only to highlight the similarity with the PIN example of [16, 7]).

Let us turn now to a more subtle kind of information leak, called *nontermination leak*, which arises in standard multi-threaded languages. In a parallel language a nontermination leak may occur when a loop on a high test is followed by a low memory change, since such a program may be temporarily blocked and then unblocked by some other thread, which then controls the time at which the low assignment is executed. Can this situation arise in a reactive setting? As already noticed, the reactive parallel operator \uparrow implements a cooperative scheduling policy. This means that once it gets the control, an instantly diverging thread will hold it forever, thus preventing the other threads from executing. Consider the typical example

$$(\text{while } x_H \neq 0 \text{ do nil}); y_L := 0 \quad (6)$$

In a standard parallel setting the loop can be unblocked by a parallel thread, thus possibly giving rise to an insecure flow between x_H and y_L . In contrast, in the reactive case the loop cannot be unblocked by another thread, since the executing thread releases the control only by terminating or suspending. However, reactive concurrency introduces new leaks which will force us to rule out programs like (6), where loops on high tests are followed by low assignments or low emissions. Consider for instance the program

$$((\text{while } x_H \neq 0 \text{ do } (\text{pause}; x_H := 0)); y_L := 0) \uparrow (y_L := 1; x_H := 0) \quad (7)$$

Here the first thread suspends if and only if $x_H \neq 0$. If it suspends, the second thread takes over and the low variable y_L gets the value 1 before the value 0. If it does not suspend, the low variable y_L gets the value 0 before the value 1. Hence there is an insecure flow from x_H to y_L . This kind of leak will be called *suspension leak*, conveying the idea that high tests may influence the suspension of threads and thus their order of execution, possibly leading to insecure flows if these threads contain low assignments or emissions.

To sum up, although the phenomena involved are slightly different, the rule of thumb for typing reactive programs seems to be similar to that used for parallel programs in [17, 7], which prescribes that *high tests*, i.e. tests on high variables or signals, should not be followed (whether in the same construct or in sequence) by *low writes*, i.e. assignments to low variables or emissions of low signals.

However, there is a further element to consider. Let us look at a more elaborate example, which obeys the above-mentioned condition of having “no low writes after high tests” and yet exhibits a

suspension leak. Consider the program $(\gamma' \uparrow \alpha') \uparrow \beta'$, running in two different signal environments $E_1 = \{a_H, b_H\}$ and $E_2 = \{b_H\}$:

$$\begin{aligned} & \gamma' : \text{pause}; x_L := 1 \\ \alpha' : & \text{do (when } a_H \text{ do nil) watching } b_H \uparrow \text{when } c_L \text{ do } x_L := 0 \\ & \beta' : \text{pause}; \text{emit } c_L \end{aligned} \quad (8)$$

Here threads γ' and α' contain different assignments to x_L . Thread γ' starts executing and suspends after a few steps. Now thread α' may either suspend immediately, in the environment E_2 where signal a_H is absent, or execute its left branch before suspending, in the environment E_1 where signal a_H is present. Therefore γ' and α' will switch positions in the environment E_1 but not in the environment E_2 . In any case their composition will eventually suspend and thread β' will gain the control, suspending as well after a few moves. At this point a change of instant occurs, after which the system is either in the state $\text{emit } c_L \uparrow (\text{when } c_L \text{ do } x_L := 0 \uparrow x_L := 1)$ or in the state $\text{emit } c_L \uparrow (x_L := 1 \uparrow (\text{nil} \uparrow \text{when } c_L \text{ do } x_L := 0))$, depending on whether the starting environment was E_1 or E_2 . In any case signal c_L is now emitted and we are left with either $(\text{when } c_L \text{ do } x_L := 0 \uparrow x_L := 1)$ or $(x_L := 1 \uparrow (\text{nil} \uparrow \text{when } c_L \text{ do } x_L := 0))$. In the first case the assignment $x_L := 0$ will be executed first, while in the second it will be executed after $x_L := 1$.

The last example shows that suspension leaks may be caused by the coexistence of a high test in a thread with a low write in another thread. This will lead us to impose conditions in the typing rules for reactive concurrency, which are similar to those required for sequential composition in [7, 17], demanding that the level of tests in one component be lower or equal to the level of writes in the other component. Moreover in the case of $P \uparrow Q$ this will have to hold in both directions, since the roles of P and Q may be interchanged during execution.

Let us now present our type system. As we mentioned in Section 2, expressions will be typed with *simple types*, which are just security levels δ, θ, σ . As usual, these are assumed to form a lattice (\mathcal{T}, \leq) , where the order relation \leq stands for “less secret than” and \wedge, \vee denote the meet and join operations. Starting from simple types we build *variable types* of the form $\delta \text{ var}$ and *signal types* of the form $\delta \text{ sig}$. Program types will have the form $(\theta, \sigma) \text{ cmd}$, as in [7, 17]. Here the first component θ represents a lower bound on the level of written variables and emitted signals, while the second component σ is an upper bound on the level of tested variables and signals.

Our type system is presented in Figure 5. Concerning the imperative part of the language, it is the same as that of [7, 17]. The rules for the reactive constructs have been mostly motivated by the above examples. Let us just note that the rules for the `when` and `watching` commands are similar to those for the `while` command. This is not surprising since all these commands consist of the execution of a process under a guard. As concerns reactive parallel composition, we already explained the reasons for introducing side conditions similar to those for sequential composition.

One may notice that these side conditions restrict the compositionality of the type system and introduce some overhead (two comparisons of security levels) when adding new threads in the system. This is the price to pay for allowing loops with high guards such as `while $x_H = 0$ do nil` (which are rejected by previous type systems, as [16, 15]) in the context of a co-routine mechanism. However, it might be worth examining if this restriction could be lifted to some extent by means of techniques proposed for other concurrent languages ([12, 13]).

$$\begin{array}{l}
\text{(NIL)} \quad \Gamma \vdash \text{nil} : (\theta, \sigma) \text{ cmd} \\
\text{(ASSIGN)} \quad \frac{\Gamma \vdash e : \theta \quad \Gamma(x) = \theta \text{ var}}{\Gamma \vdash x := e : (\theta, \sigma) \text{ cmd}} \\
\text{(LET)} \quad \frac{\Gamma \vdash e : \delta \quad \{x : \delta \text{ var}\} \Gamma \vdash P : (\theta, \sigma) \text{ cmd}}{\Gamma \vdash \text{let } x : \delta = e \text{ in } P : (\theta, \sigma) \text{ cmd}} \\
\text{(SEQ)} \quad \frac{\Gamma \vdash P : (\theta_1, \sigma_1) \text{ cmd} \quad \Gamma \vdash Q : (\theta_2, \sigma_2) \text{ cmd} \quad \sigma_1 \leq \theta_2}{\Gamma \vdash P ; Q : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}} \\
\text{(COND)} \quad \frac{\Gamma \vdash e : \delta \quad \Gamma \vdash P : (\theta, \sigma) \text{ cmd} \quad \Gamma \vdash Q : (\theta, \sigma) \text{ cmd} \quad \delta \leq \theta}{\Gamma \vdash \text{if } e \text{ then } P \text{ else } Q : (\theta, \delta \vee \sigma) \text{ cmd}} \\
\text{(WHILE)} \quad \frac{\Gamma \vdash e : \delta \quad \Gamma \vdash P : (\theta, \sigma) \text{ cmd} \quad \delta \vee \sigma \leq \theta}{\Gamma \vdash \text{while } e \text{ do } P : (\theta, \delta \vee \sigma) \text{ cmd}} \\
\text{(EMIT)} \quad \frac{\Gamma(a) = \theta \text{ sig}}{\Gamma \vdash \text{emit } a : (\theta, \sigma) \text{ cmd}} \\
\text{(LOCAL)} \quad \frac{\{a : \delta \text{ sig}\} \Gamma \vdash P : (\theta, \sigma) \text{ cmd}}{\Gamma \vdash \text{local } a : \delta \text{ in } P : (\theta, \sigma) \text{ cmd}} \\
\text{(WATCH)} \quad \frac{\Gamma(a) = \delta \text{ sig} \quad \Gamma \vdash P : (\theta, \sigma) \text{ cmd} \quad \delta \leq \theta}{\Gamma \vdash \text{do } P \text{ watching } a : (\theta, \delta \vee \sigma) \text{ cmd}} \\
\text{(WHEN)} \quad \frac{\Gamma(a) = \delta \text{ sig} \quad \Gamma \vdash P : (\theta, \sigma) \text{ cmd} \quad \delta \leq \theta}{\Gamma \vdash \text{when } a \text{ do } P : (\theta, \delta \vee \sigma) \text{ cmd}} \\
\text{(PAR)} \quad \frac{\Gamma \vdash P : (\theta_1, \sigma_1) \text{ cmd} \quad \Gamma \vdash Q : (\theta_2, \sigma_2) \text{ cmd} \quad \sigma_1 \leq \theta_2 \quad \sigma_2 \leq \theta_1}{\Gamma \vdash P \uparrow Q : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}} \\
\text{(SUB)} \quad \frac{\Gamma \vdash P : (\theta, \sigma) \text{ cmd} \quad \theta \geq \theta' \quad \sigma \leq \sigma'}{\Gamma \vdash P : (\theta', \sigma') \text{ cmd}} \\
\text{(EXPR)} \quad \frac{\forall x_i \in \text{fv}(e). \delta \geq \theta_i \text{ where } \Gamma(x_i) = \theta_i \text{ var}}{\Gamma \vdash e : \delta}
\end{array}$$

Figure 5: Typing Rules

3.2 Properties of typed programs

It is easy to see that if a program is typable in a type-environment Γ , it is also typable with the same type in any environment Γ' extending Γ . This fact, together with a simple property of substitution, is stated here without proof:

Proposition 3.1 (Simple properties of typed programs)

1. If $\Gamma \vdash P : (\theta, s) \text{ cmd}$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash P : (\theta, s) \text{ cmd}$.
2. If $\{n : \delta \text{ name}\} \Gamma \vdash P : (\theta, s) \text{ cmd}$ and $n' \notin \text{dom}(\Gamma)$, then $\{n' : \delta \text{ name}\} \Gamma \vdash \{n'/n\}P : (\theta, s) \text{ cmd}$.

In order to establish one of the main properties of our type system, subject reduction, we start by showing that types are preserved by instant changes:

Lemma 3.2 *If $\langle E, P \rangle \dagger$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$, then $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$.*

Proof By induction on the proof of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$. We only have to consider the cases of suspendable processes (see Figure 4), corresponding to the typing rules (WHEN), (WATCH), (SEQ), (PAR) and to the subtyping rule (SUB).

- (WHEN) Here $P = \text{when } a \text{ do } P_1$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma(a) = \delta \text{ sig}$, $\Gamma \vdash P_1 : (\theta, \sigma_1) \text{ cmd}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma_1$. If $a \notin E$ we conclude immediately since $[P]_E = P$. If $a \in E$ then $[P]_E = \text{when } a \text{ do } [P_1]_E$. By induction $\Gamma \vdash [P_1]_E : (\theta, \sigma_1) \text{ cmd}$, hence, using rule (WHEN) again, we deduce $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$.
- (WATCH) Here $P = \text{do } P_1 \text{ watching } a$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced again from $\Gamma(a) = \delta \text{ sig}$, $\Gamma \vdash P_1 : (\theta, \sigma_1) \text{ cmd}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma_1$. If $a \in E$ we have $[P]_E = \text{nil}$ and we can conclude immediately by rule (NIL). If $a \notin E$ then $[P]_E = \text{do } [P_1]_E \text{ watching } a$. By induction $\Gamma \vdash [P_1]_E : (\theta, \sigma_1) \text{ cmd}$, hence $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$ by rule (WATCH).
- (SEQ) Here $P = P_1 ; P_2$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1) \text{ cmd}$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$. We have $[P_1 ; P_2]_E = [P_1]_E ; P_2$. By induction $\Gamma \vdash [P_1]_E : (\theta_1, \sigma_1) \text{ cmd}$, hence $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$ by rule (SEQ).
- (PAR) Here $P = P_1 \uparrow P_2$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1) \text{ cmd}$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$, $\sigma_2 \leq \theta_1$. We have $[P_1 \uparrow P_2]_E = [P_1]_E \uparrow [P_2]_E$. By induction $\Gamma \vdash [P_1]_E : (\theta_1, \sigma_1) \text{ cmd}$ and $\Gamma \vdash [P_2]_E : (\theta_2, \sigma_2) \text{ cmd}$, hence $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$ by rule (PAR).
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta \geq \theta'$ and $\sigma \leq \sigma'$. By induction $\Gamma \vdash [P]_E : (\theta', \sigma') \text{ cmd}$, hence, using rule (SUB) again, we deduce $\Gamma \vdash [P]_E : (\theta, \sigma) \text{ cmd}$.

□

Theorem 3.3 (Subject Reduction)

If $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ and $\langle \Gamma, S, E, P \rangle \mapsto \langle \Gamma', S', E', P' \rangle$ then $\Gamma' \vdash P' : (\theta, \sigma) \text{ cmd}$.

Proof Let $C = \langle \Gamma, S, E, P \rangle$ and $C' = \langle \Gamma', S', E', P' \rangle$. We want to show that $\Gamma' \vdash P' : (\theta, \sigma) \text{ cmd}$. We distinguish the two cases $C \hookrightarrow C'$ and $C \rightarrow C'$.

1. *Instant change.* If $C \hookrightarrow C'$ then $\Gamma' = \Gamma$ and $P' = \lfloor P \rfloor_E$. We can then conclude immediately using Lemma 3.2.
2. *Simple move.* Suppose now $C \rightarrow C'$. We show that $\Gamma' \vdash P' : (\theta, \sigma) \text{ cmd}$ by induction on the proof of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$. We examine the cases where P is not terminated nor suspended. Note that in the cases (ASSIGN) and (EMIT) we have $P' = \text{nil}$ and thus we can conclude immediately using rule (NIL). We consider the other cases.
 - (LET) Here $P = \text{let } x : \delta = e \text{ in } P_1$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash e : \delta$ and $\{x : \delta \text{ var}\} \Gamma \vdash P_1 : (\theta, \sigma) \text{ cmd}$. Since $C \rightarrow C'$ is derived by (LET-OP), we have $\Gamma' = \{x' : \delta \text{ var}\} \Gamma$ and $P' = \{x'/x\} P_1$ for some $x' \notin \text{dom}(\Gamma)$. Then by Proposition 3.1 we can conclude that $\{x' : \delta \text{ var}\} \Gamma \vdash \{x'/x\} P_1 : (\theta, \sigma) \text{ cmd}$.
 - (SEQ) Here $P = P_1 ; P_2$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash P_1 : (\theta_1, \sigma_1) \text{ cmd}$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$.
 - If $P_1 = \text{nil}$, then $C \rightarrow C'$ is derived by (SEQ-OP₁) and thus $\Gamma' = \Gamma$ and $P' = P_2$. Since $\theta_1 \wedge \theta_2 \leq \theta_2$ and $\sigma_1 \vee \sigma_2 \geq \sigma_2$, by rule (SUB) we have $\Gamma' \vdash P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}$.
 - If $P_1 \neq \text{nil}$ then $C \rightarrow C'$ is derived using rule (SEQ-OP₂) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. We then have $P' = P'_1 ; P_2$. By induction $\Gamma' \vdash P'_1 : (\theta_1, \sigma_1) \text{ cmd}$. By Proposition 2.1 $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$, hence we may use Proposition 3.1 to get $\Gamma' \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$. Then, using rule (SEQ) again, we obtain $\Gamma' \vdash P'_1 ; P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}$.
 - (COND) Here $P = \text{if } e \text{ then } P_1 \text{ else } P_2$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash e : \delta$, $\Gamma \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\Gamma \vdash P_2 : (\theta, \sigma') \text{ cmd}$ where $\sigma = \delta \vee \sigma'$.
 - If $S(e) = \text{true}$, then $C \rightarrow C'$ is derived using rule (COND-OP₁) and we have $\Gamma' = \Gamma$ and $P' = P_1$. Since $\Gamma' \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\sigma \geq \sigma'$, by (SUB) we have $\Gamma' \vdash P_1 : (\theta, \sigma) \text{ cmd}$.
 - The case $S(e) = \text{false}$ is symmetric.
 - (WHILE) Here $P = \text{while } e \text{ do } P_1$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash e : \delta$, $\Gamma \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\delta \vee \sigma' \leq \theta$ where $\sigma = \delta \vee \sigma'$.
 - If $S(e) = \text{true}$, then $C \rightarrow C'$ is derived using rule (WHILE-OP₁) and we have $\Gamma' = \Gamma$ and $P' = P_1 ; \text{while } e \text{ do } P_1$. Since $\Gamma \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\sigma \geq \sigma'$, by (SUB) we have $\Gamma \vdash P_1 : (\theta, \sigma) \text{ cmd}$. Since $\sigma \leq \theta$, we may then use (SEQ) to deduce $\Gamma' \vdash P' : (\theta, \sigma) \text{ cmd}$.

- If $S(e) = \text{false}$, then $C \rightarrow C'$ is derived using rule (WHILE-OP₂). Then $\Gamma' = \Gamma$ and $P' = \text{nil}$, and we may conclude immediately by rule (NIL).
- (LOCAL) Here $P = \text{local } a : \delta \text{ in } P_1$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\{a : \delta \text{ sig}\} \Gamma \vdash P_1 : (\theta, \sigma) \text{ cmd}$. The transition $C \rightarrow C'$ is derived by rule (LOCAL-OP), thus $\Gamma' = \{a' : \delta \text{ sig}\} \Gamma$ and $P' = \{a'/a\} P_1$ for some $a' \notin \text{dom}(\Gamma)$. Then we may use Proposition 3.1 to deduce $\{a' : \delta \text{ sig}\} \Gamma \vdash \{a'/a\} P_1 : (\theta, \sigma) \text{ cmd}$.
- (WATCH) Here $P = \text{do } P_1 \text{ watching } a$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma(a) = \delta \text{ sig}, \Gamma \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\delta \leq \theta$, where $\sigma = \delta \vee \sigma'$.
 - If $P_1 = \text{nil}$, then $C \rightarrow C'$ is derived using rule (WATCH-OP₁). Then $\Gamma' = \Gamma$ and $P' = \text{nil}$, and we conclude immediately by rule (NIL).
 - If $P_1 \neq \text{nil}$, then $C \rightarrow C'$ is derived using rule (WATCH-OP₂) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. In this case $P' = \text{do } P'_1 \text{ watching } a$. By induction $\Gamma' \vdash P'_1 : (\theta, \sigma') \text{ cmd}$, so using (WATCH) again we conclude that $\Gamma' \vdash \text{do } P'_1 \text{ watching } a : (\theta, \sigma) \text{ cmd}$.
- (WHEN) Here $P = \text{when } a \text{ do } P_1$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma(a) = \delta \text{ sig}, \Gamma \vdash P_1 : (\theta, \sigma') \text{ cmd}$ and $\delta \leq \theta$, where $\sigma = \delta \vee \sigma'$.
 - If $P_1 = \text{nil}$, then $C \rightarrow C'$ is derived using rule (WHEN-OP₁). Then $\Gamma' = \Gamma$ and $P' = \text{nil}$, and we conclude immediately by rule (NIL).
 - If $P_1 \neq \text{nil}$, then $C \rightarrow C'$ is derived using rule (WHEN-OP₂) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. In this case $P' = \text{when } a \text{ do } P'_1$. By induction $\Gamma' \vdash P'_1 : (\theta, \sigma') \text{ cmd}$, so using (WHEN) again we may conclude that $\Gamma' \vdash \text{when } P'_1 \text{ do } a : (\theta, \sigma) \text{ cmd}$.
- (PAR) Here $P = P_1 \curlywedge P_2$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1) \text{ cmd}, \Gamma \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}, \theta = \theta_1 \wedge \theta_2, \sigma = \sigma_1 \vee \sigma_2, \sigma_1 \leq \theta_2, \sigma_2 \leq \theta_1$.
 - If $P_1 = \text{nil}$, then $C \rightarrow C'$ is derived using rule (PAR-OP₁). Then $\Gamma' = \Gamma$ and $P' = P_2$. Since $\theta_1 \wedge \theta_2 \leq \theta_2$ and $\sigma_1 \vee \sigma_2 \geq \sigma_2$, by rule (SUB) we have $\Gamma' \vdash P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}$.
 - If $P_1 \neq \text{nil}$ and $\neg \langle E, P \rangle_{1\ddagger}$ then $C \rightarrow C'$ is derived using rule (PAR-OP₂) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \rightarrow \langle \Gamma', S', E', P'_1 \rangle$. We then have $P' = P'_1 \curlywedge P_2$. By induction $\Gamma' \vdash P'_1 : (\theta_1, \sigma_1) \text{ cmd}$. By Proposition 2.1 $\text{dom}(\Gamma) \subseteq \text{dom}(\Gamma')$, hence we may use Proposition 3.1 to get $\Gamma' \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$. Then, using rule (PAR) again, we obtain $\Gamma' \vdash P'_1 \curlywedge P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2) \text{ cmd}$.
 - If $P_1 \neq \text{nil}$ and $\langle E, P \rangle_{1\ddagger}$ then $C \rightarrow C'$ is derived using rule (PAR-OP₃) and $C' = \langle \Gamma, S, E, P_2 \curlywedge P_1 \rangle$. In this case we can immediately conclude using rule (PAR), since this is symmetric with respect to the two components P_1 and P_2 .
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta \geq \theta'$ and $\sigma \leq \sigma'$. By induction $\Gamma' \vdash P' : (\theta', \sigma') \text{ cmd}$, hence, using rule (SUB) again, we deduce $\Gamma' \vdash P' : (\theta, \sigma) \text{ cmd}$.

□

Our next result ensures that program types have the intended meaning. We introduce first some terminology. The generic term “guard” will be used for any variable or signal appearing in a test. For instance, x is a guard in the program `(while $x \leq y$ do $y := y - 1$)` and a is a guard in the program `(when a do $x := 0$)`. Moreover, if $\Gamma \vdash P : (\theta, \sigma)$ *cmd*, we say that a variable x (resp. a signal a) of P has *security level* δ in Γ if we are in one of two cases:

- i*) x (resp. a) is free in P and Γ contains the pair $(x, \delta \text{ var})$ (resp. $(a, \delta \text{ sig})$).
- ii*) x (resp. a) is bound in P by a declaration `let $x : \delta = e$ in Q` (resp. `local $a : \delta$ in Q`).

Note that for case *ii*) the environment Γ is actually immaterial, whereas we implicitly assume that $x \notin \text{bn}(Q)$ (resp. $a \notin \text{bn}(Q)$). This condition can always be met by renaming bound variables.

Lemma 3.4 (Guard Safety and Confinement)

1. If $\Gamma \vdash P : (\theta, \sigma)$ *cmd* then every guard in P has security level $\delta \leq \sigma$.
2. If $\Gamma \vdash P : (\theta, \sigma)$ *cmd* then every written variable or emitted signal in P has security level δ such that $\theta \leq \delta$.

Proof *Proof of 1.* By induction on the inference of $\Gamma \vdash P : (\theta, \sigma)$ *cmd*.

- (NIL), (ASSIGN), (SEQ), (COND), (WHILE), (SUB): All these cases correspond to imperative constructs. The proof is the same as in [7] and is therefore omitted.
- (LET) Here $P = \text{let } x : \delta = e \text{ in } Q$, with $\{x : \delta \text{ var}\} \Gamma \vdash Q : (\theta, \sigma)$ *cmd* and $\Gamma \vdash e : \delta$. By induction every guard in Q has security level $\delta' \leq \sigma$ in the type environment $\{x : \delta \text{ var}\} \Gamma$. Then every guard of P different from x has security level $\delta' \leq \sigma$ in the type environment Γ . As for x , it has the security level δ given by its declaration, and if it appears as a guard in P that's because it appears as a (free) guard in Q , in which case we know by induction that $\delta \leq \sigma$.
- (EMIT) Vacuous, since $P = \text{emit } a$ contains no guard.
- (LOCAL) Analogous to (LET).
- (WATCH) Here $P = \text{do } Q \text{ watching } a$, with $\Gamma(a) = \delta \text{ sig}$, $\Gamma \vdash Q : (\theta, \sigma')$ *cmd* and $\sigma = \delta \vee \sigma'$. By induction every guard in Q has security level $\delta' \leq \sigma'$, and therefore $\delta' \leq \delta \vee \sigma' = \sigma$. Hence the guard a introduced by the watch construct, which has security level δ , satisfies the constraint $\delta \leq \sigma$.
- (WHEN) Analogous to (WATCH).
- (PAR) Here $P = P_1 \frown P_2$ with $\Gamma \vdash P_1 : (\theta_1, \sigma_1)$ *cmd*, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)$ *cmd* and $\sigma = \sigma_1 \vee \sigma_2$. By induction every guard in P_i has type $\delta_i \leq \sigma_i$. Since $\sigma_i \leq \sigma_1 \vee \sigma_2$ we can then conclude.

Proof of 2. By induction on the inference of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$.

- (NIL), (ASSIGN), (SEQ), (COND), (WHILE), (SUB) The proof is as in [7].
- (LET) Here $P = \text{let } x : \delta = e \text{ in } Q$, with $\{x : \delta \text{ var}\} \Gamma \vdash Q : (\theta, \sigma) \text{ cmd}$ and $\Gamma \vdash e : \delta$. By induction every written variable or emitted signal in Q has security level δ' such that $\theta \leq \delta'$ in the type environment $\{x : \delta \text{ var}\} \Gamma$. Then every written variable or emitted signal different from x in P has security level δ' such that $\theta \leq \delta'$ in the type environment Γ . The bound variable x has the security level δ given by its declaration. In case x is written in Q , we know by induction that $\theta \leq \delta$.
- (LOCAL) Analogous to (LET).
- (WATCH) Here $P = \text{do } Q \text{ watching } a$, with $\Gamma(a) = \delta \text{ sig}$ and $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$. By induction every written variable or emitted signal in Q has security level δ' such that $\theta \leq \delta'$. Whence the conclusion, since P does not introduce any written variables nor emitted signals.
- (WHEN) Analogous to (WATCH).
- (EMIT) Here $P = \text{emit } a$, and $\Gamma(a) = \theta \text{ sig}$. This case is trivial since the only emitted signal has security level θ .
- (PAR) Here $P = P_1 \uparrow P_2$ with $\Gamma \vdash P_1 : (\theta_1, \sigma_1) \text{ cmd}$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2) \text{ cmd}$ and $\theta = \theta_1 \wedge \theta_2$. By induction every written variable or emitted signal in P_i has security level δ_i with $\theta_i \leq \delta_i$. Since $\theta_1 \wedge \theta_2 \leq \theta_i$ we can then conclude.

□

3.3 Security notion

In this section we introduce our security notion and formalise it as a kind of bisimulation, which we call *reactive bisimulation*. We start by introducing some terminology that will be useful to define our notion of indistinguishability. We use \mathcal{L} to designate a *downward-closed set of security levels*, that is a set $\mathcal{L} \subseteq \mathcal{T}$ satisfying $\theta \in \mathcal{L} \ \& \ \sigma \leq \theta \Rightarrow \sigma \in \mathcal{L}$. The *low memory* is the portion of the variable-store and signal-environment to which the type-environment associates “low security levels” (i.e. security levels in \mathcal{L}). Two memories are said to be low-equal if their low parts coincide:

Definition 3.1 (\mathcal{L}, Γ -equality of Memories and Configurations) *Let S_1, S_2 be variable stores, E_1, E_2 be signal environments, $\Gamma, \Gamma_1, \Gamma_2$ be typing environments and P_1, P_2 be programs. Then low equality $=_{\mathcal{L}}^{\Gamma}$ on stores and signal environments is defined by:*

$$S_1 =_{\mathcal{L}}^{\Gamma} S_2 \text{ if } \forall x \in \text{dom}(\Gamma). (\Gamma(x) = \theta \text{ var} \ \& \ \theta \in \mathcal{L}) \Rightarrow (x \in \text{dom}(S_1) \Leftrightarrow x \in \text{dom}(S_2)) \text{ and} \\ \text{if } x \in \text{dom}(S_i) \text{ then } S_1(x) = S_2(x)$$

$$E_1 =_{\mathcal{L}}^{\Gamma} E_2 \text{ if } \forall a \in \text{dom}(\Gamma). (\Gamma(a) = \theta \text{ sig} \ \& \ \theta \in \mathcal{L}) \Rightarrow (a \in E_1 \Leftrightarrow a \in E_2)$$

Then $=_{\mathcal{L}}^{\Gamma}$ is defined pointwise on memories: $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$ if $S_1 =_{\mathcal{L}}^{\Gamma} S_2$ and $E_1 =_{\mathcal{L}}^{\Gamma} E_2$

By extension, for configurations we let: $\langle \Gamma_1, S_1, E_1, P_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle \Gamma_2, S_2, E_2, P_2 \rangle$ if $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$.

There is a class of programs for which the security property is particularly easy to establish because of their inability to change the low store. We will refer to these as *high programs*. We shall distinguish two classes of high programs, based respectively on a syntactic and a semantic analysis.

Definition 3.2 (High Programs)

1. Syntactically high programs $\mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$ is inductively defined by: $P \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$ if:

- $P = (x := e)$ and $\Gamma(x) = \theta \text{ var} \Rightarrow \theta \notin \mathcal{L}$, or
- $P = (\text{emit } a)$ and $\Gamma(x) = \theta \text{ sig} \Rightarrow \theta \notin \mathcal{L}$, or
- $P = \text{let } x : \delta = e \text{ in } Q$ and $Q \in \mathcal{H}_{\text{syn}}^{\Gamma \cup \{x:\delta \text{ var}\}, \mathcal{L}}$, or
- $P = \text{local } a : \delta \text{ in } Q$ and $Q \in \mathcal{H}_{\text{syn}}^{\Gamma \cup \{a:\delta \text{ sig}\}, \mathcal{L}}$, or
- $P = (\text{while } e \text{ do } Q)$ or $P = (\text{when } a \text{ do } Q)$ or $P = (\text{do } Q \text{ watching } a)$, where $Q \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$, or
- $P = (P_1 ; P_2)$ or $P = (\text{if } e \text{ then } P_1 \text{ else } P_2)$ or $P = (P_1 \ \dot{\vee} \ P_2)$, where $P_i \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$ for $i = 1, 2$.

2. Semantically high programs $\mathcal{H}_{\text{sem}}^{\Gamma, \mathcal{L}}$ is coinductively defined by: $P \in \mathcal{H}_{\text{sem}}^{\Gamma, \mathcal{L}}$ implies

- $\forall S, E, \langle \Gamma, S, E, P \rangle \rightarrow \langle \Gamma', S', E', P' \rangle$ implies $\langle S, E \rangle =_{\mathcal{L}}^{\Gamma} \langle S', E' \rangle$ and $P' \in \mathcal{H}_{\text{sem}}^{\Gamma', \mathcal{L}}$, and
- $\forall S, E, \langle \Gamma, S, E, P \rangle \leftrightarrow \langle \Gamma', S', E', P' \rangle$ implies $P' \in \mathcal{H}_{\text{sem}}^{\Gamma', \mathcal{L}}$

Let us comment briefly on these definitions. The notion of syntactic highness is quite straightforward. Essentially, a program is syntactically high if it does not contain assignments to low variables or emissions of low signals. Note that $P = \text{let } x : \delta = e \text{ in } Q$ (as well as $P = \text{local } a : \delta \text{ in } Q$) is considered syntactically high even if $\delta \in \mathcal{L}$, provided Q is syntactically high in the extended typing environment. The notion of semantic highness is a little more subtle. The first clause ensures that the low memory is preserved by simple moves. Note that the comparison of memories is carried out in the starting typing environment Γ . This means that in case $\Gamma' \neq \Gamma$, the newly created variable or signal will not be taken into account in the comparison; however, since its creation turns it into a free variable or signal, it will then be considered in the following steps. For instance, assuming $\delta \in \mathcal{L}$, the program $P = \text{local } a : \delta \text{ in nil}$ is semantically high while $Q = \text{local } a : \delta \text{ in emit } a$ is not. The second clause of Definition 3.2.2 concerns instant changes. As argued in the Introduction, we do not consider as observable the initialization of the low signal environment that is induced by instant changes. This is reflected by the absence of the low equality condition in the second clause of Definition 3.2.2 (recall that the variable-store S is not modified during an instant change). Thanks to this weaker requirement at instant changes, it may be easily shown that syntactic highness implies semantic highness:

Fact 3.5 *For any Γ and for any downward-closed set \mathcal{L} of security levels, $\mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}} \subseteq \mathcal{H}_{\text{sem}}^{\Gamma, \mathcal{L}}$.*

As may be expected, the converse is not true. An example of a semantically high program that is not syntactically high is $P = \text{if true then nil else } y_L := 0$.

Clearly, both properties of syntactic and semantic highness are preserved by execution. Moreover:

Fact 3.6 *If $\exists \theta \notin \mathcal{L}, \exists \sigma$ such that $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$, then $P \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$.*

Proof Immediate, by the Confinement lemma. □

We introduce now the notion of \mathcal{L} -Guardedness, borrowed from [7]. This formalises the property that a program contains no high guards.

Definition 3.3 (\mathcal{L} -Guardedness)

A program P is \mathcal{L} -guarded in Γ if $\exists \theta, \exists \sigma \in \mathcal{L}$ such that $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$.

Fact 3.7 *If P is \mathcal{L} -guarded in Γ then every guard in P has security level $\delta \in \mathcal{L}$ in Γ .*

Proof Immediate, by the Guard Safety lemma. □

We shall sometimes use the complementary notion of *non- \mathcal{L} -guardedness* in Γ , for a program P which is typable in Γ but for which there does not exist $\sigma \in \mathcal{L}$ and θ such that $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$.

The following result says that \mathcal{L} -guarded programs, when computing over low-equal memories, produce at each step equal typing environments and programs, and low-equal memories.

Theorem 3.8 (Behaviour of \mathcal{L} -Guarded Programs)

Let P be \mathcal{L} -guarded in Γ and $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$. Then

1. (Instant change) $\langle \Gamma, S_1, E_1, P \rangle \hookrightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ implies $\langle \Gamma, S_2, E_2, P \rangle \hookrightarrow \langle \Gamma', S'_2, E'_2, P' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$.
2. (Simple moves) $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ implies $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma', S'_2, E'_2, P' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$.

Proof *Proof of 1.* By induction on the inference of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ where $\sigma \in \mathcal{L}$, and then by case analysis on the definition of $\langle E_1, P \rangle_{\ddagger}$ (Figure 1). We only have to consider suspendable processes (Figure 4), corresponding to the typing rules (WHEN), (WATCH), (SEQ), (PAR) and to the subtyping rule (SUB). Note that it is enough to show that $\langle E_1, P \rangle_{\ddagger}$ implies $\langle E_2, P \rangle_{\ddagger}$, because in this case rule (INSTANT-OP) yields the transitions $\langle \Gamma, S_1, E_1, P \rangle \hookrightarrow \langle \Gamma, S_1, \emptyset, [P]_E \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \hookrightarrow \langle \Gamma, S_2, \emptyset, [P]_E \rangle$, where $\langle S_1, \emptyset \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, \emptyset \rangle$ follows from $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$.

- (WHEN) Here $P = \text{when } a \text{ do } Q$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$, $\Gamma(a) = \delta \text{ sig}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. There are two cases to consider for $\langle E_1, P \rangle_{\ddagger}$, depending on the last rule used to prove it:
 1. $\langle E_1, P \rangle_{\ddagger}$ is deduced by rule (WHEN-SUS₁): in this case $a \notin E_1$. Note that $\delta \leq \sigma$ implies $\delta \in \mathcal{L}$ and therefore, since $E_1 =_{\mathcal{L}}^{\Gamma} E_2$, $a \in E_1 \Leftrightarrow a \in E_2$. Then also $a \notin E_2$ and we can apply rule (WHEN-SUS₁) to deduce $\langle E_2, P \rangle_{\ddagger}$.
 2. $\langle E_1, P \rangle_{\ddagger}$ is deduced by rule (WHEN-SUS₂) from the hypothesis $\langle E_1, Q \rangle_{\ddagger}$. Since $\sigma' \leq \sigma$ implies $\sigma' \in \mathcal{L}$, Q is \mathcal{L} -guarded. Then we have $\langle E_2, Q \rangle_{\ddagger}$ by induction, whence by rule (WHEN-SUS₂) we obtain $\langle E_2, P \rangle_{\ddagger}$.
- (WATCH), (SEQ), (PAR) By straightforward induction as in the second case of (WHEN).
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. Thus $\sigma' \in \mathcal{L}$ and we can conclude using induction.

Proof of 2. By induction on the inference of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ where $\sigma \in \mathcal{L}$, and by case analysis on the last rule used in this inference.

- (ASSIGN) Here $P = x := e$ with $\Gamma \vdash e : \theta$ and $\Gamma(x) = \theta \text{ var}$. By rule (ASSIGN-OP) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S'_1, E_1, \text{nil} \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S'_2, E_2, \text{nil} \rangle$, where $S'_1 = \{x \mapsto S_1(e)\}S_1$ and $S'_2 = \{x \mapsto S_2(e)\}S_2$. It is easy to see that $\langle S'_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E_2 \rangle$ since $\Gamma' = \Gamma$ and thus $E_1 =_{\mathcal{L}}^{\Gamma'} E_2$ is already known, while $S'_1 =_{\mathcal{L}}^{\Gamma'} S'_2$ follows from $S_1 =_{\mathcal{L}}^{\Gamma} S_2$ if $\theta \notin \mathcal{L}$, and from the additional fact that $S_1(e) = S_2(e)$ if $\theta \in \mathcal{L}$.

- (LET) Here $P = \text{let } x : \delta = e \text{ in } Q$, and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash e : \delta$ and $\{x : \delta \text{ var}\} \Gamma \vdash Q : (\theta, \sigma) \text{ cmd}$, where $\sigma \in \mathcal{L}$. Then $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma', S'_1, E_1, P' \rangle$ is deduced using rule (LET-OP), and for some $x' \notin \text{dom}(\Gamma)$, we have $\Gamma' = \{x' : \delta \text{ var}\} \Gamma$, $S'_1 = \{x' \mapsto S_1(e)\} S_1$ and $P' = \{x'/x\} P$. Then, using rule (LET-OP) again and choosing the same x' we obtain $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma', S'_2, E_2, P' \rangle$, where $S'_2 = \{x' \mapsto S_2(e)\} S_2$. Now $E_1 =_{\mathcal{L}}^{\Gamma'} E_2$ follows from $E_1 =_{\mathcal{L}}^{\Gamma} E_2$ and the fact that $x' \notin E_1$ and $x' \notin E_2$, while $S'_1 =_{\mathcal{L}}^{\Gamma'} S'_2$ follows, as in the previous case, from the fact that $S_1 =_{\mathcal{L}}^{\Gamma} S_2$ if $\theta \notin \mathcal{L}$, and from the additional fact that $S_1(e) = S_2(e)$ if $\theta \in \mathcal{L}$.
- (SEQ) Here $P = Q ; R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$.
 1. If $Q = \text{nil}$, then by rule (SEQ-OP₁) we have both $\langle \Gamma, S_1, E_1, Q ; R \rangle \rightarrow \langle \Gamma, S_1, E_1, R \rangle$ and $\langle \Gamma, S_2, E_2, Q ; R \rangle \rightarrow \langle \Gamma, S_2, E_2, R \rangle$, so we can conclude immediately.
 2. If $Q \neq \text{nil}$, then the transition $\langle \Gamma, S_1, E_1, Q ; R \rangle \rightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ is derived by rule (SEQ-OP₂) from the hypothesis $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$, and $P' = Q' ; R$. Since $\sigma' \leq \sigma$, Q is \mathcal{L} -guarded. Hence by induction $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. We then have $\langle \Gamma, S_2, E_2, Q ; R \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' ; R \rangle$ by (SEQ-OP₂) and we can conclude.
- (COND) Here $P = \text{if } e \text{ then } Q \text{ else } R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash e : \delta$, $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta, \sigma') \text{ cmd}$, where $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. Since P is \mathcal{L} -guarded in Γ , $\sigma \in \mathcal{L}$. Then also $\delta \in \mathcal{L}$ and therefore, since by rule (EXPR) each variable occurring in e has level less than or equal to δ , we have $S_1(e) = S_2(e)$. Now, if $S_i(e) = \text{true}$, then by rule (COND-OP₁) $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, Q \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, Q \rangle$ and we can conclude immediately. The case where $S_i(e) = \text{false}$ is symmetric.
- (WHILE) Similar to (COND).
- (EMIT) Here $P = \text{emit } a$, where $\Gamma(a) = \theta \text{ sig}$. By rule (EMIT-OP) we have for $i = 1, 2$ the transition $\langle \Gamma, S_i, E_i, \text{emit } a \rangle \rightarrow \langle \Gamma, S_i, E'_i, \text{nil} \rangle$, where $E'_i = \{a\} \cup E_i$. Then all we have to show is that $E'_1 =_{\mathcal{L}}^{\Gamma} E'_2$. If $\theta \notin \mathcal{L}$ this follows immediately from $E_1 =_{\mathcal{L}}^{\Gamma} E_2$; if $\theta \in \mathcal{L}$, it also uses the fact that $a \in E'_i$ for both i .
- (LOCAL) Analogous to (LET), using (LOCAL-OP) instead of (LET-OP) and considering signal names and environments instead of variables and stores.
- (WATCH) Here $P = \text{do } Q \text{ watching } a$, with $\Gamma(a) = \delta \text{ sig}$, $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$.
 1. $Q = \text{nil}$. In this case by rule (WATCH-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, \text{nil} \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, \text{nil} \rangle$, so we can conclude immediately.

2. $Q \neq \text{nil}$. Then $\langle \Gamma, S_1, E_1, \text{do } Q \text{ watching } a \rangle \rightarrow \langle \Gamma', S'_1, E'_1, \text{do } Q' \text{ watching } a \rangle$ is derived by rule (WATCH-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$. Note that $\sigma' \leq \sigma$ implies $\sigma' \in \mathcal{L}$, thus Q is also \mathcal{L} -guarded. Then by induction we have a transition $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Whence by (WATCH-OP₂) we deduce $\langle \Gamma, S_2, E_2, \text{do } Q \text{ watching } a \rangle \rightarrow \langle \Gamma', S'_2, E'_2, \text{do } Q' \text{ watching } a \rangle$, which is the required matching transition.
- (WHEN) Here $P = \text{when } a \text{ do } Q$, with $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$, $\Gamma(a) = \delta \text{ sig}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. As in the previous case, there are two possibilities:
 1. $Q = \text{nil}$. In this case by rule (WHEN-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, \text{nil} \rangle$. Note that $\delta \leq \sigma$ implies $\delta \in \mathcal{L}$ and therefore, since $E_1 =_{\mathcal{L}}^{\Gamma} E_2$, $a \in E_1 \Leftrightarrow a \in E_2$. We know that $a \in E_1$, thus also $a \in E_2$. We can then apply rule (WHEN-OP₁) again to get $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, \text{nil} \rangle$.
 2. $Q \neq \text{nil}$. Then $\langle \Gamma, S_1, E_1, \text{when } Q \text{ do } a \rangle \rightarrow \langle \Gamma', S'_1, E'_1, \text{when } Q' \text{ do } a \rangle$ is derived by rule (WHEN-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$. Since $\sigma' \leq \sigma$ implies $\sigma' \in \mathcal{L}$, Q is \mathcal{L} -guarded. Then by induction $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Then since $a \in E_2$ we can apply rule (WHEN-OP₂) to deduce $\langle \Gamma, S_2, E_2, \text{when } Q \text{ do } a \rangle \rightarrow \langle \Gamma', S'_2, E'_2, \text{when } Q' \text{ do } a \rangle$, and we conclude.
 - (PAR) Here $P = Q \uparrow R$ with $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$ and $\sigma = \sigma' \vee \sigma''$. Since $\sigma' \leq \sigma$ and $\sigma'' \leq \sigma$, Q and R are also \mathcal{L} -guarded. There are three possibilities:
 1. $Q = \text{nil}$. In this case by rule (PAR-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, R \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, R \rangle$, and we can conclude.
 2. $Q \neq \text{nil}$ and $\neg \langle E_1, Q \rangle \ddagger$. Then $\langle \Gamma, S_1, E_1, Q \uparrow R \rangle \rightarrow \langle \Gamma_1, S'_1, E'_1, Q' \uparrow R \rangle$ is derived using rule (PAR-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma_1, S'_1, E'_1, Q' \rangle$. By induction we have $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma_2, S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Then we may use again (PAR-OP₂) to deduce $\langle \Gamma, S_2, E_2, Q \uparrow R \rangle \rightarrow \langle \Gamma_2, S'_2, E'_2, Q' \uparrow R \rangle$.
 3. $\langle E_1, Q \rangle \ddagger$ and $\neg \langle E_1, R \rangle \ddagger$. Then $\langle \Gamma, S_1, E_1, Q \uparrow R \rangle \rightarrow \langle \Gamma, S_1, E_1, R \uparrow Q \rangle$ by (PAR-OP₃). Since Q and R are also \mathcal{L} -guarded, by Clause 1. of the theorem statement we have $\langle E_2, Q \rangle \ddagger$ and $\neg \langle E_2, R \rangle \ddagger$. Then we may apply (PAR-OP₃) again to deduce the transition $\langle \Gamma, S_2, E_2, Q \uparrow R \rangle \rightarrow \langle \Gamma, S_2, E_2, R \uparrow Q \rangle$ and we conclude.
 - (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. Thus $\sigma' \in \mathcal{L}$ and we can conclude using induction.

□

The key result for proving noninterference is the following theorem, which states that a typable non \mathcal{L} -guarded program preserves low-equality of memories as long as it encounters only low tests, and becomes syntactically high as soon as it meets a high test. Note that while \mathcal{L} -guardedness is preserved by subterms (and by execution), the complementary property, non \mathcal{L} -guardedness, is not. Therefore the next theorem will make use of Theorem 3.8.

Theorem 3.9 (Behaviour of non \mathcal{L} -Guarded Programs)

Let P be typable and non \mathcal{L} -Guarded in Γ and let $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$. Then either $P \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$ or one of the following holds:

1. (Instant change) $\langle \Gamma, S_1, E_1, P \rangle \hookrightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ implies $\langle \Gamma, S_2, E_2, P \rangle \hookrightarrow \langle \Gamma', S'_2, E'_2, P' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$.
2. (Simple moves) $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ implies $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma', S'_2, E'_2, P' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$.

Proof The proof amounts to trying to show the properties of Theorem 3.8 also for non low-guarded programs and, in case they do not hold, to show that P is syntactically high. We prove the two clauses separately, by induction on the inference of $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$.

Proof of 1. We start by showing that if $\langle E_1, P \rangle \dagger$ then either $\langle E_2, P \rangle \dagger$ (in which case the transition $\langle \Gamma, S_1, E_1, P \rangle \hookrightarrow \langle \Gamma, S_1, \emptyset, [P]_E \rangle$ is matched by $\langle \Gamma, S_2, E_2, P \rangle \hookrightarrow \langle \Gamma, S_2, \emptyset, [P]_E \rangle$ using rule (INSTANT-OP), as in the proof of Theorem 3.8), or $P \in \mathcal{H}_{\text{syn}}^{\Gamma, \mathcal{L}}$. Again, we only have to examine suspendable processes, corresponding to the typing rules (WHEN), (WATCH), (SEQ), (PAR) and to the subtyping rule (SUB).

- (WHEN) Here $P = \text{when } a \text{ do } Q$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}$, $\Gamma(a) = \delta \text{ sig}$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. There are two cases for $\langle E_1, P \rangle \dagger$:
 1. $\langle E_1, P \rangle \dagger$ is deduced by rule (WHEN-SUS₁): in this case $a \notin E_1$. Then either $a \notin E_2$ and we can conclude using rule (WHEN-SUS₁), or $a \in E_2$. In the latter case, since $E_1 =_{\mathcal{L}}^{\Gamma} E_2$, it must be $\delta \notin \mathcal{L}$. Whence, since $\delta \leq \theta$, we deduce that also $\theta \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 we conclude that P is syntactically high.
 2. $\langle E_1, P \rangle \dagger$ is deduced by rule (WHEN-SUS₂) from the hypothesis $\langle E_1, Q \rangle \dagger$. By induction either $\langle E_2, Q \rangle \dagger$, in which case also $\langle E_2, P \rangle \dagger$ by rule (WHEN-SUS₂), or Q is syntactically high. In the latter case by Definition 3.2 also P is syntactically high.
- (WATCH) Easy induction as in the second case of (WHEN).
- (SEQ) Here $P = Q ; R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$.
In this case $\langle E_1, P \rangle \dagger$ is deduced by rule (SEQ-SUS) from $\langle E_1, Q \rangle \dagger$. By induction either $\langle E_2, Q \rangle \dagger$, in which case $\langle E_2, P \rangle \dagger$ by rule (SEQ-SUS) again, or Q is syntactically high and, by virtue of Theorem 3.8, not \mathcal{L} -guarded. This means that $\sigma' \notin \mathcal{L}$ and thus, since $\sigma' \leq \theta''$, also $\theta'' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 R is syntactically high and thus, by Definition 3.2, also $Q; R$ is syntactically high.
- (PAR) Here $P = Q \uparrow R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$ and $\sigma'' \leq \theta'$.

There are three possibilities:

1. both Q and R are non \mathcal{L} -guarded. This means that both $\sigma' \notin \mathcal{L}$ and $\sigma'' \notin \mathcal{L}$ and thus, since $\sigma' \leq \theta''$ and $\sigma'' \leq \theta'$, also $\theta'' \notin \mathcal{L}$ and $\theta' \notin \mathcal{L}$. Hence by the Confinement Lemma 3.4 both Q and R are syntactically high and thus by Definition 3.2 also $P = Q \uparrow R$ is syntactically high.
 2. one of Q and R is \mathcal{L} -guarded and the other is not. Suppose Q is \mathcal{L} -guarded and R is not (the other case is symmetric). We know that $\langle E_1, P \rangle_{\dagger}$ is deduced by rule (PAR-SUS) from $\langle E_1, Q \rangle_{\dagger}$ and $\langle E_1, R \rangle_{\dagger}$. Since Q is \mathcal{L} -guarded, by Theorem 3.8 we have $\langle E_2, Q \rangle_{\dagger}$. Since R is not \mathcal{L} -guarded we know by induction that either $\langle E_2, R \rangle_{\dagger}$ or R is syntactically high. If $\langle E_2, R \rangle_{\dagger}$ we may apply rule (PAR-SUS) to get $\langle E_2, P \rangle_{\dagger}$. Otherwise we use the fact that $\sigma'' \notin \mathcal{L}$ (because R is not \mathcal{L} -guarded) and thus, since $\sigma'' \leq \theta'$, also $\theta' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 Q is syntactically high and thus also the composition $P = Q \uparrow R$ is syntactically high.
 3. both Q and R are \mathcal{L} -guarded (note that this is possible although P is non \mathcal{L} -guarded). As in the previous case, we know that $\langle E_1, Q \rangle_{\dagger}$ and $\langle E_1, R \rangle_{\dagger}$. Then by Theorem 3.8 we obtain $\langle E_2, Q \rangle_{\dagger}$ and $\langle E_2, R \rangle_{\dagger}$. Whence by (PAR-SUS) we may conclude that $\langle E_2, P \rangle_{\dagger}$.
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. Then we may conclude immediately, using Theorem 3.8 if $\sigma' \in \mathcal{L}$, and induction otherwise.

Proof of 2. We consider now the case where $\neg \langle E_1, P \rangle_{\dagger}$. For the rules (ASSIGN), (LET), (EMIT), (LOCAL) the result is proved exactly as for Theorem 3.8, since in these cases it does not depend on the hypothesis of \mathcal{L} -guardedness. We examine the remaining cases.

- (SEQ) Here $P = Q ; R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$.
 1. If $Q = \text{nil}$, then by rule (SEQ-OP₁) we have both $\langle \Gamma, S_1, E_1, Q; R \rangle \rightarrow \langle \Gamma, S_1, E_1, R \rangle$ and $\langle \Gamma, S_2, E_2, Q; R \rangle \rightarrow \langle \Gamma, S_2, E_2, R \rangle$, so we can conclude immediately.
 2. If $Q \neq \text{nil}$, then the transition $\langle \Gamma, S_1, E_1, Q; R \rangle \rightarrow \langle \Gamma', S'_1, E'_1, P' \rangle$ is derived by rule (SEQ-OP₂) from the hypothesis $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$, and $P' = Q'; R$. There are two possibilities:
 - Q is \mathcal{L} -guarded. Then by Theorem 3.8 we have $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Hence by (SEQ-OP₂) we can conclude.
 - Q is not \mathcal{L} -guarded. By induction either $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$, and we conclude using (SEQ-OP₂) as in the previous case, or Q is syntactically high. In this case, we use the fact that $\sigma' \notin \mathcal{L}$ (because Q is not \mathcal{L} -guarded) and therefore, since $\sigma' \leq \theta''$, also $\theta'' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 R is syntactically high, and thus by Definition 3.2 we conclude that $Q; R$ is syntactically high.

- (COND) Here $P = \text{if } e \text{ then } Q \text{ else } R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash e : \delta, \Gamma \vdash Q : (\theta, \sigma') \text{ cmd}, \Gamma \vdash R : (\theta, \sigma') \text{ cmd}$, where $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$.
If $\delta \in \mathcal{L}$ we have $S_1(e) = S_2(e)$ and we can conclude easily as in the proof of Theorem 3.8. Otherwise $\delta \notin \mathcal{L}$. Then, since $\delta \leq \theta$, also $\theta \notin \mathcal{L}$ and by the Confinement Lemma 3.4 both Q and R are syntactically high. Hence by Definition 3.2 also P is syntactically high.
- (WHILE) Similar to (COND).
- (WATCH) Here $P = \text{do } Q \text{ watching } a$, with $\Gamma(a) = \delta \text{ sig}, \Gamma \vdash Q : (\theta, \sigma') \text{ cmd}, \delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. We distinguish two cases:
 1. $Q = \text{nil}$. In this case by rule (WATCH-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, \text{nil} \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, \text{nil} \rangle$, so we can conclude immediately.
 2. $Q \neq \text{nil}$. Then $\langle \Gamma, S_1, E_1, \text{do } Q \text{ watching } a \rangle \rightarrow \langle \Gamma', S'_1, E'_1, \text{do } Q' \text{ watching } a \rangle$ is derived by rule (WATCH-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$. There are two subcases:
 - Q is \mathcal{L} -guarded. Then by Theorem 3.8 we have $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Then by rule (WATCH-OP₂) we get the matching transition $\langle \Gamma, S_2, E_2, \text{do } Q \text{ watching } a \rangle \rightarrow \langle \Gamma', S'_2, E'_2, \text{do } Q' \text{ watching } a \rangle$.
 - Q is not \mathcal{L} -guarded. Then by induction either $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$ and we conclude as in the previous case, or Q is syntactically high. In the latter case by Definition 3.2 also P is syntactically high.
- (WHEN) Here $P = \text{when } a \text{ do } Q$, with $\Gamma \vdash Q : (\theta, \sigma') \text{ cmd}, \Gamma(a) = \delta \text{ sig}, \delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. Assume $a \in E_1$ (otherwise this case is vacuous). There are two possibilities:
 1. $Q = \text{nil}$. Then by rule (WHEN-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, \text{nil} \rangle$. If $\delta \in \mathcal{L}$ then $a \in E_1 \Leftrightarrow a \in E_2$ and we proceed as in the proof of Theorem 3.8. If $\delta \notin \mathcal{L}$, since $\delta \leq \theta$, also $\theta \notin \mathcal{L}$. Hence by the Confinement Lemma 3.4 Q is syntactically high and thus by Definition 3.2 also P is syntactically high.
 2. $Q \neq \text{nil}$. Then $\langle \Gamma, S_1, E_1, \text{when } a \text{ do } Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, \text{when } a \text{ do } Q' \rangle$ is derived by rule (WHEN-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma', S'_1, E'_1, Q' \rangle$. There are two subcases:
 - Q is \mathcal{L} -guarded. Then by Theorem 3.8 we have $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$, with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Since $\delta \notin \mathcal{L}$ then $a \in E_2$ and we can apply rule (WHEN-OP₂) to deduce $\langle \Gamma, S_2, E_2, \text{when } Q \text{ do } a \rangle \rightarrow \langle \Gamma', S'_2, E'_2, \text{when } Q' \text{ do } a \rangle$.
 - Q is not \mathcal{L} -guarded. Then by induction either $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$, or Q is syntactically high. In the latter case we conclude immediately, by Definition 3.2, that P is syntactically high. In the former case, there are two possibilities: if $\delta \in \mathcal{L}$ then $a \in E_2$ and we can apply rule (WHEN-OP₂) as in the case where Q is \mathcal{L} -guarded; if $\delta \notin \mathcal{L}$, since $\delta \leq \theta$ also $\theta \notin \mathcal{L}$. Then by Lemma 3.4 Q is syntactically high, hence P is syntactically high.

- (PAR) Here $P = Q \dot{\wedge} R$ and $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma') \text{ cmd}$, $\Gamma \vdash R : (\theta'', \sigma'') \text{ cmd}$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$ and $\sigma'' \leq \theta'$. We distinguish two cases, depending on whether $\langle E_1, Q \rangle_{\ddagger}$ or $\neg \langle E_1, Q \rangle_{\ddagger}$. Assume first that $\neg \langle E_1, Q \rangle_{\ddagger}$. There are two possibilities :
 1. $Q = \text{nil}$. In this case by rule (PAR-OP₁) we have $\langle \Gamma, S_1, E_1, P \rangle \rightarrow \langle \Gamma, S_1, E_1, R \rangle$ and $\langle \Gamma, S_2, E_2, P \rangle \rightarrow \langle \Gamma, S_2, E_2, R \rangle$, and we can conclude.
 2. $Q \neq \text{nil}$. Then $\langle \Gamma, S_1, E_1, Q \dot{\wedge} R \rangle \rightarrow \langle \Gamma_1, S'_1, E'_1, Q' \dot{\wedge} R \rangle$ is derived by (PAR-OP₂) from $\langle \Gamma, S_1, E_1, Q \rangle \rightarrow \langle \Gamma_1, S'_1, E'_1, Q' \rangle$. There are two subcases:
 - Q is \mathcal{L} -guarded. Then by Theorem 3.8 we have $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$. Hence by (PAR-OP₂) we can conclude immediately.
 - Q is not \mathcal{L} -guarded. By induction either $\langle \Gamma, S_2, E_2, Q \rangle \rightarrow \langle \Gamma', S'_2, E'_2, Q' \rangle$ with $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'} \langle S'_2, E'_2 \rangle$, and we conclude using (PAR-OP₂) as in the previous case, or Q is syntactically high. In the latter case, we use the fact that Q is not \mathcal{L} -guarded to deduce that $\sigma' \notin \mathcal{L}$ and therefore, since $\sigma' \leq \theta''$, also $\theta'' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 R is syntactically high, and thus by Definition 3.2 also $Q; R$ is syntactically high.

Consider now the case where $\langle E_1, Q \rangle_{\ddagger}$. In this case it must be $\neg \langle E_1, R \rangle_{\ddagger}$ and by (PAR-OP₃) $\langle \Gamma, S_1, E_1, Q \dot{\wedge} R \rangle \rightarrow \langle \Gamma, S_1, E_1, R \dot{\wedge} Q \rangle$. Here there are four possibilities:

1. Q and R are both non \mathcal{L} -guarded. Then, as in the corresponding case of Part 1, we deduce that both Q and R are syntactically high and thus also $P = Q \dot{\wedge} R$ is syntactically high.
 2. Q is \mathcal{L} -guarded and R is not. In this case we may use Theorem 3.8 to obtain $\langle E_2, Q \rangle_{\ddagger}$. Since R is not \mathcal{L} -guarded we know by induction that either $\neg \langle E_2, R \rangle_{\ddagger}$ or R is syntactically high. If $\neg \langle E_2, R \rangle_{\ddagger}$ we may use rule (PAR-OP₃) to conclude. Otherwise we use the fact that $\sigma'' \notin \mathcal{L}$ (because R is not \mathcal{L} -guarded) and thus, since $\sigma'' \leq \theta'$, also $\theta' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 Q is syntactically high and thus also $P = Q \dot{\wedge} R$ is syntactically high.
 3. R is \mathcal{L} -guarded and Q is not. By Theorem 3.8 we have that $\neg \langle E_2, R \rangle_{\ddagger}$. By induction we know that either $\langle E_2, Q \rangle_{\ddagger}$ or Q is syntactically high. If $\langle E_2, Q \rangle_{\ddagger}$ we use rule (PAR-OP₃) to conclude. Otherwise we use the fact that $\sigma' \notin \mathcal{L}$ (because Q is not \mathcal{L} -guarded) and thus, since $\sigma' \leq \theta''$, also $\theta'' \notin \mathcal{L}$. Then by the Confinement Lemma 3.4 we know that R is syntactically high and thus also $P = Q \dot{\wedge} R$ is syntactically high.
 4. Q and R are both \mathcal{L} -guarded. In this case by Theorem 3.8 we have $\langle E_2, Q \rangle_{\ddagger}$ and $\neg \langle E_2, R \rangle_{\ddagger}$, and we conclude immediately using rule (PAR-OP₃).
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma) \text{ cmd}$ is deduced from $\Gamma \vdash P : (\theta', \sigma') \text{ cmd}$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. We may then conclude immediately, using Theorem 3.8 if $\sigma' \in \mathcal{L}$, and induction otherwise.

□

We are now ready to define our notion of security for programs. This will be formalised as usual as a kind of self-bisimulation: a program is secure if it behaves in the same way in all low-equivalent memories. In fact our bisimulation is slightly non-standard, in that it factors out high programs instead of requiring them to behave in the same way on low-equal memories. This can be explained as follows. Recall that in reactive computations all signals are reset to absent at the beginning of each instant. This means that the low signal environment is not preserved by instant changes unless it is empty. As a consequence, two high programs resulting from a fork after a high test will not in general have the same effect on the low signal environment, since one of them may jump to the next instant while the other one does not. On the other hand we know that a semantically high program preserves the low store (cf Def. 3.2). Indeed, in the next section we will show that our reactive notion of security implies a more standard one, which only requires the low store to be preserved.

Definition 3.4 (Reactive \mathcal{L} -Bisimulation) *The partial equivalence $\sim_{\mathcal{L}}$ is the largest symmetric relation \mathcal{R} on configurations such that $C_1 = \langle \Gamma_1, S_1, E_1, P_1 \rangle \mathcal{R} \langle \Gamma_2, S_2, E_2, P_2 \rangle = C_2$ implies $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} \langle S_2, E_2 \rangle$ and one of the following properties, where $C'_i = \langle \Gamma'_i, S'_i, E'_i, P'_i \rangle$:*

1. $P_i \in \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$, or
2. $C_1 \hookrightarrow C'_1$ implies $C_2 \hookrightarrow C'_2$ with $C'_1 \mathcal{R} C'_2$, or
3. $\langle \Gamma_1, S_1, E_1, P_1 \rangle \rightarrow \langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle$ with¹ $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$ implies $\langle \Gamma_2, S_2, E_2, P_2 \rangle \rightarrow \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$ with¹ $n \in \text{dom}(\Gamma'_2 - \Gamma_2) \Rightarrow n \in \text{dom}(\Gamma'_1 - \Gamma_1)$ and $\langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle \mathcal{R} \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$.

Some further comments will be helpful. As explained above, as soon as two programs become semantically high in low-equal memories, they are immediately $\sim_{\mathcal{L}}$ -related by Clause 1, without further verification. Note that this separation between high and “low” programs allows us to use strong bisimulation requirements in Clauses 2 and 3, which would have to be weakened if they had to apply also to high programs. Moreover the two conditions on new names in Clause 3 ensure that, when playing the bisimulation game on two programs, one does not fail to equate or distinguish them for bad reasons, to do with the choice of new names. To this end, the first program should choose new names which are not free in the second program, and the second program should mimic the choice of new names of the first.

Let us illustrate more precisely the use of these conditions with a couple of examples. Consider the program $P = (\text{let } x : L = 0 \text{ in } y_L := x)$. Note that P is not semantically high. Then, without the first condition of Clause 3 we would have $C_1 = \langle \Gamma_1, S_1, \emptyset, P \rangle \not\sim_{\mathcal{L}} \langle \Gamma_2, S_2, \emptyset, P \rangle = C_2$, where $\Gamma_1 : y_L \mapsto L$, $\Gamma_2 : x \mapsto L, y_L \mapsto L$ and S_1, S_2 are the two low-equal stores $S_1 : y_L \mapsto 0$, $S_2 : x \mapsto 1, y_L \mapsto 0$. Indeed, if C_1 was allowed to choose x as its new name, then C_2 would not be able to respond by picking the same name because $x \in \text{dom}(\Gamma_2)$, and if C_2 were allowed to pick a different name x' (supposing the second condition was not there to forbid it), then the resulting store S'_2 would not be low-equal to S'_1 since it would give a different value

¹these two conditions on new names are not necessary for our soundness result, but make sense for arbitrary configurations and render our security notion stronger, as will be explained below.

to x . Note that the pair of configurations (C_1, C_2) is reachable² by running the program $Q = \text{if } z_H = 0 \text{ then } (\text{nil}; P) \text{ else } (\text{let } x : L = 1 \text{ in } P)$ in the identical typing environments $\bar{\Gamma}_i : y_L \mapsto L, z_H \mapsto H$ and in any pair of low-equal stores \bar{S}_1, \bar{S}_2 such that $\bar{S}_1(z_H) = 0$ and $\bar{S}_2(z_H) \neq 0$. Although Q is not typable, it seems reasonable to consider it secure since the two branches have the same effect on the low memory. To sum up, the first condition ensures that local and global names are not confused and that configurations are not distinguished by accident. This condition is always satisfiable since the set of names is countable.

As for the second condition of Clause 3, it ensures that the security notion properly takes into account local names. Consider for instance the programs $P_1 = (\text{let } x : L = 0 \text{ in } x := x + 1)$ and $P_2 = (\text{let } x : L = 1 \text{ in } x := x + 1)$, which only differ for the initial value of the local name. Without the second condition we would have $C_1 = \langle \emptyset, \emptyset, \emptyset, P_1 \rangle \sim_{\mathcal{L}} \langle \emptyset, \emptyset, \emptyset, P_2 \rangle = C_2$, because in response to the choice of a local name x_1 by C_1 , a different local name x_2 could be chosen by C_2 and the resulting memories would be trivially equivalent because their domains are disjoint. In other words, without the second condition we could possibly elude the comparison of “local memories”. On the contrary, we take here the stand that local memories should be part of what is observable by a possibly malicious party. Then, the possibility that P_1 and P_2 act differently on the local store should appear, and the way to enforce it is to require that the second process chooses exactly the same local name as the first. Indeed, with the second condition we have $C_1 \not\sim_{\mathcal{L}} C_2$. Note that, as in the previous example, the pair of configurations (C_1, C_2) is reachable², by running the (not semantically high) program $Q = (\text{if } z_H = 0 \text{ then } P_1 \text{ else } P_2)$ in the identical typing environments $\bar{\Gamma}_i : z_H \mapsto H$ and in any pair of low-equal stores \bar{S}_1, \bar{S}_2 such that $\bar{S}_1(z_H) = 0$ and $\bar{S}_2(z_H) \neq 0$. Symmetrically to the first condition, the second condition ensures that configurations are not equated by accident.

The set of *secure programs* is now defined to be the reflexive kernel of $\sim_{\mathcal{L}}$, namely the set of programs which are bisimilar to themselves in any two low-equivalent memories:

Definition 3.5 (Γ -Secure Programs) *P is secure in Γ if for any downward-closed set \mathcal{L} of security levels and for any S_i, E_i such that $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$ we have $\langle \Gamma, S_1, E_1, P \rangle \sim_{\mathcal{L}} \langle \Gamma, S_2, E_2, P \rangle$.*

As a matter of fact, since the environment Γ is part of our configurations and, in the comparison of a program with itself in Definition 3.5, it is required to be the same in the two initial configurations, it will turn out that the two conditions of Clause 3 are not necessary to prove our soundness result. Indeed, for that purpose Clause 3 will only be applied to pairs of configurations C_1, C_2 such that $\Gamma_1 = \Gamma_2$ and $P_1 = P_2$, which evolve by Theorems 3.8 and 3.9 to configurations C'_1, C'_2 such that $\Gamma'_1 = \Gamma'_2$ and $P'_1 = P'_2$, thus trivially satisfying the two conditions on new names. However these conditions make sense when comparing arbitrary configurations, as shown by the examples above, and the first one will be necessary for proving Theorem 3.11.

²up to the addition of the global variable z_H to both the Γ_i 's and the S_i 's

3.4 Soundness of the type system

In this section we establish our soundness result, namely we prove that every typable program is secure. This result rests heavily on the Theorems 3.8 and 3.9 proved in the previous section, which describe the one-step behaviour of typable programs (respectively low-guarded and non low-guarded). We also introduce a more standard notion of bisimulation, and prove that it is strictly weaker than reactive bisimulation, which can then be seen as a refinement of the usual notion.

Theorem 3.10 (Typability \Rightarrow Noninterference) *If P is typable in Γ then P is Γ -secure.*

Proof For any downward-closed \mathcal{L} , define the relation $\mathcal{S}_{\mathcal{L}}$ on configurations as follows:

$C_1 = \langle \Gamma_1, S_1, E_1, P_1 \rangle \mathcal{S}_{\mathcal{L}} \langle \Gamma_2, S_2, E_2, P_2 \rangle = C_2$ if and only if P_i is typable in Γ_i for $i = 1, 2$, and

- $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} \langle S_2, E_2 \rangle$
- and one of the following holds:
 1. $P_i \in \mathcal{H}_{\text{syn}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$, or
 2. $\langle \Gamma_1, P_1 \rangle = \langle \Gamma_2, P_2 \rangle$.

Note first that if P is typable in Γ and $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$, then $\langle \Gamma, S_1, E_1, P \rangle \mathcal{S}_{\mathcal{L}} \langle \Gamma, S_2, E_2, P \rangle$ by Clause 2. We prove now that $\mathcal{S}_{\mathcal{L}} \subseteq \sim_{\mathcal{L}}$ by showing that $\mathcal{S}_{\mathcal{L}}$ is a $\sim_{\mathcal{L}}$ -bisimulation. Suppose $C_1 \mathcal{S}_{\mathcal{L}} C_2$. Then $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} \langle S_2, E_2 \rangle$ and we are in one of two cases:

1. $P_i \in \mathcal{H}_{\text{syn}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$, by Clause 1. Since $\mathcal{H}_{\text{syn}}^{\Gamma_i, \mathcal{L}} \subseteq \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}$ we have then $P_i \in \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}$ and therefore $C_1 \sim_{\mathcal{L}} C_2$ by Clause 1 of Def. 3.4.
2. $\langle \Gamma_1, P_1 \rangle = \langle \Gamma_2, P_2 \rangle$, by Clause 2. We may assume $P_i \notin \mathcal{H}_{\text{syn}}^{\Gamma_i, \mathcal{L}}$, since otherwise we would fall back in the previous case. Suppose $C_1 \hookrightarrow C'_1$ (respectively, $C_1 \rightarrow C'_1$), where $C'_1 = \langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle$. Then, using Theorem 3.8 or Theorem 3.9 depending on whether P_1 is low-guarded or not (note that in the latter case we also use the fact that $P_1 \notin \mathcal{H}_{\text{syn}}^{\Gamma_1, \mathcal{L}}$) we may deduce $C_2 \hookrightarrow C'_2$ (respectively, $C_2 \rightarrow C'_2$), for some $C'_2 = \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$ such that $\langle \Gamma'_1, P'_1 \rangle = \langle \Gamma'_2, P'_2 \rangle$ and $\langle S'_1, E'_1 \rangle =_{\mathcal{L}}^{\Gamma'_1 \cap \Gamma'_2} \langle S'_2, E'_2 \rangle$. Note that in the case where $C_1 \rightarrow C'_1$ is matched by $C_2 \rightarrow C'_2$, the condition $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$ is satisfied because $\Gamma_2 = \Gamma_1$, and the condition $n \in \text{dom}(\Gamma'_2 - \Gamma_2) \Rightarrow n \in \text{dom}(\Gamma'_1 - \Gamma_1)$ is satisfied because additionally $\Gamma'_2 = \Gamma'_1$. In all cases we have $C'_1 \mathcal{S}_{\mathcal{L}} C'_2$ and we may conclude.

□

To show that our approach “conservatively extends” previous ones, we shall now turn to a different notion of security, based on a more usual kind of bisimulation where programs are only required to preserve the low store and high programs are not distinguished from the others. As a counterpart, some of the observation power on local names (including variables) will be lost.

Definition 3.6 (Weak reactive \mathcal{L} -Bisimulation) *The partial equivalence $\simeq_{\mathcal{L}}$ is the largest symmetric relation \mathcal{R} on configurations such that $\langle \Gamma_1, S_1, E_1, P_1 \rangle \mathcal{R} \langle \Gamma_2, S_2, E_2, P_2 \rangle$ implies:*

- $S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$ and
- $\langle \Gamma_1, S_1, E_1, P_1 \rangle \mapsto \langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle$ with $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$ implies $\langle \Gamma_2, S_2, E_2, P_2 \rangle \mapsto^* \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$ with $\langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle \mathcal{R} \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$.

Note that the second condition on new names of Definition 3.4 does not appear here. Indeed, the requirement that new names should be chosen in the same way on both sides would be neutralized in Definition 3.6 by the possibility that a move be simulated by the empty move. To see this, let us look back at the programs P_1 and P_2 defined at page 33. Clearly, the second condition of Definition 3.4 would not help us distinguish the two configurations $C_1 = \langle \emptyset, \emptyset, \emptyset, P_1 \rangle$ and $C_2 = \langle \emptyset, \emptyset, \emptyset, P_2 \rangle$. Indeed, in response to the choice of a local name x_1 by C_1 , C_2 could idle for one turn and then choose a different local name x_2 at the next step. Thus $C_1 \simeq_{\mathcal{L}} C_2$, with or without the condition. This example suggests that $\simeq_{\mathcal{L}}$ is weaker than $\sim_{\mathcal{L}}$ not only because it ignores signals, but also because, by treating in a uniform way high and “low” programs, it is less constraining on the latter.

As for the first condition, it may be justified by the same example (C_1, C_2) used for $\sim_{\mathcal{L}}$ at page 32, since without this condition C_1 could choose x as its new name and C_2 would not be able to respond, neither by picking the same name, since $x \in \text{dom}(\Gamma_2)$, nor by idling since $S'_1(x) \neq S_2(x)$.

Associated with $\simeq_{\mathcal{L}}$ we have a new notion of security:

Definition 3.7 (Γ -Weakly Secure Programs) *P is weakly secure in Γ if for any downward-closed \mathcal{L} and for any S_i, E_i such that $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2 \rangle$ we have $\langle \Gamma, S_1, E_1, P \rangle \simeq_{\mathcal{L}} \langle \Gamma, S_2, E_2, P \rangle$.*

We show now that reactive bisimulation $\sim_{\mathcal{L}}$ is strictly included into weak reactive bisimulation $\simeq_{\mathcal{L}}$. To see that $\simeq_{\mathcal{L}} \not\subseteq \sim_{\mathcal{L}}$ consider the program $P = (\text{if } x_H = 0 \text{ then emit } a_L \text{ else emit } b_L)$. Clearly, if Γ is the typing environment specified by the subscripts and S_1, S_2 are stores such that $S_1(x_H) = 0$ and $S_2(x_H) \neq 0$, then $\langle \Gamma, S_1, \emptyset, P \rangle \simeq_{\mathcal{L}} \langle \Gamma, S_2, \emptyset, P \rangle$ but $\langle \Gamma, S_1, \emptyset, P \rangle \not\sim_{\mathcal{L}} \langle \Gamma, S_2, \emptyset, P \rangle$.

Theorem 3.11 ($\sim_{\mathcal{L}}$ is a refinement of $\simeq_{\mathcal{L}}$)

Let \mathcal{L} be a downward-closed set of security levels. Then $\sim_{\mathcal{L}} \subseteq \simeq_{\mathcal{L}}$.

Proof Define the relation $\mathcal{R}_{\mathcal{H}}$ as follows:

$$\mathcal{R}_{\mathcal{H}} = \{(C_1, C_2) \mid C_i = \langle \Gamma_i, S_i, E_i, P_i \rangle, P_i \in \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}, S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2\}$$

We show that $\mathcal{R} = \sim_{\mathcal{L}} \cup \mathcal{R}_{\mathcal{H}}$ is a $\simeq_{\mathcal{L}}$ -bisimulation. Let $C_i = \langle \Gamma_i, S_i, E_i, P_i \rangle$ and assume first that $C_1 \sim_{\mathcal{L}} C_2$. Then $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} \langle S_2, E_2 \rangle$, and thus the condition $S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$ is satisfied. Suppose now $C_1 \mapsto C'_1 = \langle \Gamma'_1, S'_1, E'_1, P'_1 \rangle$, with $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$. There are two cases to consider:

1. $P_i \in \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$. We distinguish two subcases, depending on whether the transition is a simple move or an instant change:
 - $C_1 \rightarrow C'_1$. Then by Definition 3.2 we have $P'_1 \in \mathcal{H}_{\text{sem}}^{\Gamma'_1, \mathcal{L}}$, with $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1} \langle S'_1, E'_1 \rangle$. Correspondingly we can choose $C_2 \rightarrow^* C'_2 = \langle \Gamma'_2, S'_2, E'_2, P'_2 \rangle$, with $C'_2 = C_2$ and thus $\Gamma'_2 = \Gamma_2$, $S'_2 = S_2$, $E'_2 = E_2$ and $P'_2 = P_2$. Then $P'_2 \in \mathcal{H}_{\text{sem}}^{\Gamma_2, \mathcal{L}}$ and what is left to show is that $S'_1 =_{\mathcal{L}}^{\Gamma'_1 \cap \Gamma_2} S_2$. Since $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$, we have $\Gamma'_1 \cap \Gamma_2 = \Gamma_1 \cap \Gamma_2$. Moreover, since $x \in \text{dom}(S'_1 - S_1)$ implies $x \notin \text{dom}(\Gamma_1)$ (because of the condition in the operational rule (LET)) and a fortiori $x \notin \text{dom}(\Gamma_1 \cap \Gamma_2)$, the property $S'_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$ follows from $S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$. We conclude that $C'_1 \mathcal{R}_{\mathcal{H}} C'_2$.
 - $C_1 \hookrightarrow C'_1$. Again, by Definition 3.2 we have $P'_1 \in \mathcal{H}_{\text{sem}}^{\Gamma'_1, \mathcal{L}}$. Moreover, since the transition is deduced by rule (INSTANT-OP), we know that $\Gamma'_1 = \Gamma_1$, $S'_1 = S_1$ and $E'_1 = \emptyset$. Correspondingly we choose again the transition $C_2 \mapsto^* C'_2$. Then, as in the previous case, $P'_2 \in \mathcal{H}_{\text{sem}}^{\Gamma_2, \mathcal{L}}$ and what is left to show is that $S'_1 =_{\mathcal{L}}^{\Gamma'_1 \cap \Gamma_2} S_2$. But this follows immediately from $S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$, since $S'_1 = S_1$ and $\Gamma'_1 = \Gamma_1$. Hence $C'_1 \mathcal{R}_{\mathcal{H}} C'_2$.
2. $P_i \notin \mathcal{H}_{\text{sem}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$. Then, if the transition is of the form $C_1 \hookrightarrow C'_1$ we know by Clause 2 of Definition 3.4 that there is a matching transition $C_2 \hookrightarrow C'_2$ with $C'_1 \sim_{\mathcal{L}} C'_2$. Similarly, if the transition is of the form $C_1 \rightarrow C'_1$, with $n \in \text{dom}(\Gamma'_1 - \Gamma_1) \Rightarrow n \notin \text{dom}(\Gamma_2)$, we know by Clause 3 of Def. 3.4 that $C_2 \rightarrow C'_2$, with $C'_1 \sim_{\mathcal{L}} C'_2$.

Assume now that $C_1 \mathcal{R}_{\mathcal{H}} C_2$. This case is handled in exactly the same way as Case 1 above, since the additional hypothesis of Case 1, namely $E_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} E_2$, is not used in its proof. \square

A natural question to ask now is whether the two partial equivalence relations (and therefore the two security notions) coincide on the subset of imperative programs. It turns out that this is not the case. Consider $P = \text{if } x_H = 0 \text{ then } (\text{let } u : L = 0 \text{ in } u := 0) \text{ else } (\text{let } v : L = 1 \text{ in } v := 1)$. This program is not secure (note that it is not semantically high) but it is weakly secure because if one branch picks a new name, the other can idle for one turn and then choose a different name. Instead, $Q = \text{if } x_H = 0 \text{ then } (\text{let } u : L = 0 \text{ in } z_L := u) \text{ else } (\text{let } v : L = 1 \text{ in } z_L := v)$ is neither secure nor weakly secure, since it may assign different values to the low global variable z_L .

Indeed, there are three reasons why $\sim_{\mathcal{L}}$ is stronger than $\simeq_{\mathcal{L}}$. The first is that $\sim_{\mathcal{L}}$ looks at the signal environment while $\simeq_{\mathcal{L}}$ does not. This difference of course disappears on the subset of imperative programs. The second reason is that $\simeq_{\mathcal{L}}$ allows a move to be simulated by the empty move, thus relaxing the matching requirement on new names, as illustrated by the example at page 35 or by the above one. The third reason is that the definition of $\sim_{\mathcal{L}}$ refers to a quite strong notion of semantic highness, requiring a high program to be tested in all pairs of low-equal memories at each step of its execution. Instead, in the definition of $\simeq_{\mathcal{L}}$ the quantification on memories is made once and for all at the beginning of the execution, for all programs. Consider the program $R = \text{let } y : L = 0 \text{ in } (\text{if } x_H = 0 \text{ then } y := 0 \text{ else nil})$. We let the reader verify that R is not secure while it is weakly secure (in the environment Γ giving type H to x_H).

4 Conclusion and related work

In this paper we have addressed the question of noninterference for reactive programs. We have presented a type system guaranteeing noninterference in a core imperative reactive language. We are currently studying a call-by-value language for mobility built around a reactive core, called ULM [5]. For this purpose, we intend to put together the techniques developed here with the work of [1] on the non-disclosure policy. We are working on a generalization of non-disclosure to the imperative core of ULM that should be suitable to account for the mobility fragment of the language.

As has been observed, reactive programs obey a fixed scheduling policy, which is enforced here in a syntactic way by means of the parallel construct \uparrow . Other approaches to noninterference in the presence of scheduling include the probabilistic one, proposed for instance in [18] and [15]. In these papers scheduling is introduced at the semantic level (adding probabilities to the transitions), and security is formalized through a notion of probabilistic noninterference. It should be noted that, unlike [7], which allows to express different scheduling policies, and [15] which accounts for an arbitrary scheduler (satisfying some reasonable properties), here the scheduling is fixed. Indeed, the novelty of our work resides mainly in addressing the question of noninterference in a reactive scenario. The work [14] examines the impact of synchronization on information flow, and uses it as a means to study time leaks without explicitly introducing a scheduler. However the analogy between [14] and our work cannot be pushed very far since [14] has no notion of instant (and thus no way of recovering from deadlocks) and uses asynchronous parallel composition.

Acknowledgments

We would like to thank David Sands for the suggestion of proving a refinement result for our reactive bisimulation with respect to a more standard bisimulation. The first author would further like to thank Andrei Sabelfeld for stimulating discussions during her visit at Chalmers University of Technology. This work was partially funded by the EU IST FET Project MIKADO, by the french ACI Project CRISS, and by the PhD scholarship POSI/SFRH/BD/7100/2001.

References

- [1] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy, 2005. To be presented at the *Computer Security Foundations Workshop CSFW05*, Aix-en-Provence.
- [2] A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In *Workshop on Foundations of Computer Security FCS04*, 2004. June 2004, Turku, Finland. TUCS Report n. 31.
- [3] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *IEEE*, 91(1):64–83, 2003.
- [4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comput. Programming*, 19:87–152, 1992.

-
- [5] G. Boudol. ULM, a core programming model for global computing. In *ESOP'04*, 2004.
 - [6] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *ICALP'01*, number 2076 in Lecture Notes in Computer Science, pages 382–395, 2001.
 - [7] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.
 - [8] F. Boussinot. Reactive Programming, Software. URL: <http://www-sop.inria.fr/mimosa/rp/>, 2003.
 - [9] J.-F. Susini F. Boussinot. The sugarcubes tool box: a reactive JAVA framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.
 - [10] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
 - [11] N. Halbwachs. Synchronous programming of reactive systems, 1993.
 - [12] N. Yoshida K. Honda. A uniform type structure for secure information flow. In *Proceedings of the The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, 2002.
 - [13] A. Myers S. Zdancewic. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, 2003.
 - [14] A. Sabelfeld. The impact of synchronization on secure information flow in concurrent programs. In *Proceedings of Andrei Ershov 4th International Conference on Perspectives of System Informatics*, 2001.
 - [15] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In IEEE, editor, *13th Computer Security Foundations Workshop*, 2000.
 - [16] 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.
 - [17] Geoffrey Smith. A new type system for secure information flow. In ACM, editor, *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.
 - [18] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3), 1999.
 - [19] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.



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

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)

<http://www.inria.fr>

ISSN 0249-6399