Typing Noninterference for Reactive Programs ⋆

Ana Almeida Matos [a,1] Gérard Boudol [a] Ilaria Castellani [a,*]

[a] *INRIA, 2004 Route des Lucioles, 06902 Sophia Antipolis, France*

Abstract

We study the security property of *noninterference* for a class of synchronous programs called *reactive programs*. We consider a core reactive language, obtained by extending the imperative language of Volpano, Smith and Irvine with a form of scheduled parallelism and with reactive primitives that manipulate broadcast signals. The definition of noninterference has to be tuned to the particular nature of reactive computations, which are regulated by a notion of instant. Moreover, a new form of covert channel may arise in reactive computations, called *suspension leak*. We give a formulation of noninterference based on bisimulation, as is now usual for concurrent languages. We then propose a type system to enforce this property in our language. Our type system is inspired by that introduced by Boudol and Castellani, and independently by Smith, for a parallel language with scheduling. We establish the soundness of our type system with respect to our new notion of noninterference. We finally show that this notion of noninterference refines in several aspects the standard one for imperative languages.

Key words: Reactive programming, noninterference, bisimulation, type systems.

1 Introduction

Controlling information flow in computing systems has been a long-standing problem. It has recently acquired new relevance with the advent and deployment of mobile code technology. Indeed, in order to be fully exploitable, this

⋆ Work partially funded by the European IST FET Project MIKADO, by the French ACI Project CRISS, and by the Portuguese PhD scholarship POSI/SFRH/BD/7100/2001.

* Corresponding author.

[1] Present address: Departamento de Engenharia Informática, Instituto Superior Técnico, Tagus Park, 2780-990 Porto Salvo, Portugal.

technology should provide formal guarantees about the security of the flow of information that takes place between a mobile program and its hosting environment. For instance, foreign code should not be allowed to corrupt, nor to disclose, secret data owned by its execution context: the first property is usually referred to as *integrity* and the second as *confidentiality*.

In a recent paper [11], we introduced a core programming model for mobile code called ULM, advocating the use of a *locally synchronous* programming style [9] in a *globally asynchronous* computing context. We argued that such a combination of synchronous and asynchronous paradigms could be appropriate to deal with the characteristics of mobile programming in large or "global" networks. For instance, in such networks the failure of a node may be indistinguishable from its absence of response within a given time, and synchronous programming provides time-out mechanisms that allow programs to control their waiting time when trying to communicate with a distant node.

This paper is a step towards the analysis of security issues in the ULM programming model. We concentrate here on confidentiality, and more specifically on the *noninterference* property, for a simplified version of the ULM language that does not include its mobility fragment nor its functional fragment. We simply deal with the imperative and reactive parts of the language, and with their combination. A study of noninterference and *nondisclosure* (a generalisation of noninterference allowing for declassification) has now been carried out in [4] also for the functional kernel of ULM, extended with dynamic thread creation and a declassification construct, but not including reactive constructs. A similar study has then been proposed for an extension of this language with mobility in [2,3]. The next step will be to combine all these contributions into a global treatment of information flow for the whole language ULM.

Let us start by recalling the main features of the *synchronous programming style* [9], in its control-oriented incarnation:

Broadcast signals Program components react to the presence or absence of signals, by computing and emitting signals that are broadcast to all components of the same "synchronous area".

Suspension Program components may be in a suspended state, because they are waiting for a signal which is absent at the moment when 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 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 e.g. [17,15]. It differs from the synchronous language ESTEREL [10], for instance in the way it handles the absence of signals: in reactive programming, the absence of a signal can only be determined at the end of an instant, and reaction is postponed to the following instant. In this way, one can avoid the causal paradoxes that arise in ESTEREL, making reactive programming well suited for systems where concurrent components may be dynamically added or removed, as 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 Volpano, Smith and Irvine [27] 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 shall give a formulation of noninterference based on bisimulation, as is now standard for concurrent languages [24,22,23,13]. We will then define a type system to enforce this property, along the lines of that proposed by Boudol and Castellani [14], and independently by Smith [23], for a parallel language with scheduling.

Let us briefly recall the intuition about noninterference (referring the reader to [21] for a complete survey). The idea is that in a system with multiple security levels, information should only be allowed to flow from lower to higher levels [18]. Security levels are usually assumed to form a lattice. In most of our examples we shall use only two security levels, *low* (public) and *high* (secret). In our language security levels will be attributed to both variables and signals, and we will often use subscripts to specify them: for instance x_H will denote a variable of high level and a_L a signal of low 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 structure of control. A typical example is the program

$$\texttt{if } x_H = 0 \texttt{ then } y_L := 0 \texttt{ else } y_L := 1 \tag{1}$$

where the final value of y_L may give information about the initial value of x_H. Moreover, programs may be considered secure or not depending on the context in which they appear. For instance, the program

$$(\texttt{while } x_H \neq 0 \texttt{ do nil}) \; ; \; y_L := 0 \tag{2}$$

may be viewed as safe in a sequential setting (as for instance, in the lan-

guage of [27]), since whenever it terminates it produces the same value for y_L [2]. However this program becomes unsafe in the presence of asynchronous parallelism, as is now well known (see e.g. [24,22,23,13,14]). To pinpoint the problem, let us examine the program $(P \parallel Q)$, where $\parallel$ denotes *asynchronous parallel composition* (interleaving), and P and Q are given by:

$$\begin{array}{l} P \,:\, (\texttt{while}\ x_H \neq 0\ \texttt{do nil})\ ;\ y_L := 0\ ;\ x_H := 1 \\ Q \,:\, (\texttt{while}\ x_H = 0\ \texttt{do nil})\ ;\ y_L := 1\ ;\ x_H := 0 \end{array} \tag{3}$$

Here the low variable y_L will be given different final values depending on the initial value of x_H. Hence the whole program is insecure. This means that in a concurrent setting the components P and Q should themselves be viewed as insecure. In a type system designed to enforce security, such programs can be ruled out by requiring that *high loops* (loops with a high condition) should never be followed by *low assignments* (assignments to low variables). This solution was proposed in [23,14]. A more drastic solution, forbidding altogether the use of high loops, had been previously suggested in [24] and adopted by a number of authors, e.g. [22].

Similar examples can be given to show that the use of *high conditionals* should also be restricted. In [24,22], this restriction consists in forbidding loops in the branches of high conditionals. On the other hand, in [23,14], the requirement proposed for high conditionals is similar to that for high loops, namely, none of these statements should be followed by low assignments. This choice results in a type system where program types have two components, the first representing a lower bound on the security level of assigned variables (this component coincides with the usual type for programs in security type systems) and the second an upper bound on the security level of *guards* (conditions). Then, in order to type the sequential composition $(P\,;\,Q)$, where P has type (θ_1, σ_1) and Q has type (θ_2, σ_2), one must make sure that the condition $\sigma_1 \leq \theta_2$ is satisfied. Clearly, programs P and Q above violate this condition. Importantly, this type system is robust with respect to the introduction of scheduling, as shown in [23,14].

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

[2] The notion of noninterference we consider here for sequential programs is the same as that adopted in [27], namely *termination insensitive* noninterference (we follow here the terminology of [21]), which is not able to distinguish termination from nontermination and simply ignores nonterminating computations since they do not produce a final value.

(1) The *global asynchronous composition* of "reactive machines" (synchronous areas): this is similar to the parallel composition usually considered in the literature, 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 way.
(2) The *local synchronous composition* of threads within each reactive machine. As in the implementation of reactive programming [15], we assume a deterministic *cooperative scheduling discipline* on threads. It is well known that scheduling introduces new possibilities of information flow (see e.g. [24,22,14]), and this will indeed be the case with the scheduling that we adopt here. Not surprisingly, we shall need a type system tailored for scheduling, similar to that of [23,14].

Let us illustrate with some examples the use of reactive constructs and the security leaks they can generate. Reactive constructs allow programs to emit signals, create local signals, and suspend their execution while waiting for a signal to be emitted by a concurrent program. Furthermore, the *synchronous parallel operator*, denoted $\curlywedge$, is quite different from the asynchronous parallel operator $\|$ (interleaving) that is usually adopted in concurrent languages. Indeed, while $\|$ corresponds to a fully nondeterministic or *preemptive scheduling*, $\curlywedge$ imposes a deterministic *cooperative scheduling* discipline on reactive programs. More precisely, $(P \curlywedge Q)$ executes P until it terminates or suspends, and then gives the control to Q, which similarly executes until termination or suspension; then, since Q may have emitted signals that unblock P, the control is given back to P, and so on in an alternating fashion, until both components are terminated or suspended.

Programs are executed in the context of a *memory*, which consists of a variable store, as usual, and of a *signal environment* which records the signals that have been emitted within an instant. Besides cooperative scheduling, the main characteristics of reactive computations are *suspension*, *preemption* and the subdivision of execution into *instants*. We examine each of these features in turn.

Suspension

Suspension is introduced by the construct ($\texttt{when}\ a\ \texttt{do}\ P$), whose behaviour is to execute P when signal a is present in the signal environment and suspend its execution otherwise. Consider the program:

$$\texttt{when}\ a_H\ \texttt{do}\ y_L := 0 \tag{4}$$

This program sets y_L to 0 if a_H is present in the signal environment, and suspends otherwise. It could be seen as a reactive counterpart of program (1).

A reactive analogue of program (2) could be:

$$(\texttt{when}\ a_H\ \texttt{do nil})\,;\ y_L := 0 \tag{5}$$

However, the analogy is not completely stringent since *suspension* is a specific program status, which lies somewhere in between normal termination and nontermination: like nontermination, it prevents the rest of the program from being executed; like termination, it causes the control to be handed to another component or, if all parallel components are terminated or suspended, it provokes a transition to the following instant (as will be explained shortly). Let us see a couple of examples illustrating the difference between suspension and nontermination.

Suppose we compose program (5) with a program that simply emits a_H:

$$((\texttt{when}\ a_H\ \texttt{do nil})\,;\ y_L := 0) \,\triangleleft\, \texttt{emit}\ a_H \tag{6}$$

Then the resulting program is secure. Indeed, no matter whether a_H is present or not in the initial signal environment, the low assignment will always take place: if a_H is initially present then the first component exits the `when` statement and executes the assignment, hence the second component gets the control and emits a_H; if a_H is initially absent then the first component suspends, the second component takes over and emits a_H, and finally the first component can perform the assignment. From the standpoint of a *low observer*, having access only to the low part of the memory, there is no difference between the two cases. The same holds if we replace program (5) by (4) in this example.

On the other hand, if we compose program (2) with a program that sets variable x_H to 0:

$$((\texttt{while}\ \ x_H \neq 0\ \ \texttt{do nil})\ ;\ y_L := 0) \,\triangleleft\, x_H := 0 \tag{7}$$

then the situation is quite different. Here, if x_H is initially equal to 0, then the loop is immediately exited and the low assignment is performed, before the second component gets the control. Instead, if x_H is initially different from 0, the first component loops forever and the second component never gets the control (note the difference with $\|$, which would allow the second component to unblock the first). This brings up an important aspect of cooperative scheduling: whereas it ensures the atomicity of thread computations (since a thread cannot be interrupted unless it decides to), to work properly this type of scheduling requires each thread to "cooperate" and yield the control after a finite number of steps. Conditions that guarantee the absence of divergence within an instant (and even bound the length of admissible computations within an instant, as well as their memory consumption), have been recently examined in the literature [8,7]. We shall not be concerned with this question here.

Preemption

A crucial feature of reactive programs is *preemption*, provided by the construct (`do` P `watching` a), whose behaviour is to execute P until the end of the current instant and then to abort if signal a is present, while staying unchanged otherwise. Indeed, the two constructs `when` and `watching` take their full significance when used in combination with each other. Consider the following program, which includes (4) as a subprogram:

$$\texttt{emit}\ c_L\,;\ \texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ y_L := 0)\ \texttt{watching}\ c_L \tag{8}$$

Whether a_H is present or not, this program always terminates, because the `watching` construct is killed at the end of the instant (if not already terminated). However the low assignment is performed only if a_H is present. Hence this program is insecure, and therefore so is program (4). Note that if we plug program (5) instead of (4) in this example, we obtain essentially the same behaviour. This means that program (5) is insecure too. Indeed, as a consequence of cooperative scheduling, programs (4) and (5) are equivalent in any reactive context. Note that this would not be the case if the asynchronous parallel operator $\|$ (amounting to preemptive scheduling) was used instead.

So far we have seen simple examples of reactive programs, having the same flavour as those for imperative programs, and calling for similar restrictions in the type system. A larger, practical example will be given in Section 3, once the language has been formally introduced. There is, in fact, an additional issue to face when considering the security of reactive programs, associated with the passage of instants.

Instants

A fundamental characteristic of the reactive model is the passage from an instant to the next. An *instant* is an interval of computation where all threads execute up to termination or suspension, and reach a consistent view of the presence or absence of signals. An *instant change* occurs when, possibly after several rounds of execution, all threads become inactive (either terminated or suspended). One of the effects of an instant change is to reset all signals to "absent". Another effect is to kill all `watching` commands whose controlling signal is present, thus unblocking some of the suspended threads. Let us consider an example, which builds on Example (8) above, except that the low assignment is now performed after the `watching` statement, and dependent on the presence of c_L:

$$\texttt{emit}\ c_L\,;\ \texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ c_L\,;\ \texttt{when}\ c_L\ \texttt{do}\ y_L := 0 \tag{9}$$

Here, in case a_H is present, the first `when` statement terminates and so does the `watching` statement, then giving control to the second `when` statement. Since c_L has been emitted, the low assignment can be performed. On the other hand, if a_H is absent then the program suspends on the first `when` statement, and since there are no other threads, the instant terminates. At this point, the body of the `watching` statement is killed because signal c_L is present, and a new instant starts, where the signal c_L is absent. In this case the second `when` statement suspends and the assignment does not take place.

Example (9) shows how instant changes may prevent some behaviours, as a consequence of the reset of signals. On the other side, instant changes may also allow new behaviours, by deleting parts of programs that are suspended. Consider for instance the program, obtained from (9) by composing a new thread to the right of the second `when` statement:

$$\begin{array}{l} \texttt{emit}\ c_L\ ;\ \texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ c_L\ ; \\ ((\texttt{when}\ c_L\ \texttt{do}\ y_L := 0)\ \leftcurvearrow\ (\texttt{emit}\ c_L\ ;\ y_L := 1)) \end{array} \tag{10}$$

Note that this program always terminates: if a_H is present it terminates in the first instant executing $y_L := 0$ and then $y_L := 1$, while if a_H is absent it suspends at the first instant and terminates at the second instant executing $y_L := 1$ before $y_L := 0$. Thus the order in which the assignments are executed depends on the occurrence or not of an instant change. Information leaks caused by suspension, as those of Examples (8), (9), (10), will be called *suspension leaks*.

Instant changes may be programmed. Indeed, with the constructs of our language we are able to write, for any security level, a program `pause`, whose behaviour is to suspend for the current instant, and then terminate (this program, which makes use of the local signal declaration, will be described in detail in Section 2.2). With the help of `pause` we may write the following:

$$\texttt{emit}\ a_L\ ;\ \texttt{if}\ x_H = 0\ \texttt{then}\ \texttt{nil}\ \texttt{else}\ \texttt{pause} \tag{11}$$

This program starts by emitting signal a_L. Then, depending on the value of x_H, it either terminates within an instant, in which case a_L remains present, or suspends and changes instant, in which case a_L is withdrawn. However, since instant changes are not statically predictable (they may be implemented in a variety of ways), it is not possible to rule out programs such as (11) by means of a type system. Indeed, such programs will be considered safe according to our security notion. This is because the reset of low signals at instant change will not be observable per se, but only if it is followed by changes in the low memory as in Examples (9) and (10).

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. We then proceed to define noninterference and prove the soundness of our type system. We finally compare our security notion with a more standard one, and prove that ours is stronger in various respects. This paper is the full version of [5], completed with proofs, with an additional result and with more elaborate examples.

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 boolean and arithmetic expressions, ranged over by e, e', is obtained by applying the usual total operations to constants and variables. We shall not detail this set further here, as it is entirely standard. 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.

Definition 2.1 (Reactive language) *The language of reactive programs (or processes, or threads) $P, Q \in Proc$ is defined by:*

$$P ::= \texttt{nil} \mid x := e \mid \texttt{let}\ x : \delta = e\ \texttt{in}\ P \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \mid \texttt{while}\ e\ \texttt{do}\ P \mid$$

$$P\,;Q \mid \texttt{emit}\ a \mid \texttt{local}\ a : \delta\ \texttt{in}\ P \mid \texttt{do}\ P\ \texttt{watching}\ a \mid \texttt{when}\ a\ \texttt{do}\ P \mid (P \curlywedge Q)$$

Note the use of brackets to make explicit the precedence of $\curlywedge$ (which, as we shall see, is a non associative operator). The construct $\texttt{let}\ x : \delta = e\ \texttt{in}\ P$ binds free occurrences of variable x in P. Similarly, $\texttt{local}\ a : \delta\ \texttt{in}\ P$ binds free occurrences of signal a in P. The free variables and signals of a program P, denoted by $\texttt{fv}(P)$ and $\texttt{fs}(P)$ respectively, are defined in the usual way, as well as the bound variables and signals of P, denoted by $\texttt{bv}(P)$ and $\texttt{bs}(P)$. The types of local names must be specified in the local declaration constructs because, as we shall see, the type environment only assigns types to free names. We shall use the following abbreviations: $\texttt{await}\ a =_{\text{def}} (\texttt{when}\ a\ \texttt{do nil})$ and $\texttt{loop}\ P =_{\text{def}} (\texttt{while true do}\ P)$. Most of the language constructs have been informally described in the Introduction. We give now their formal semantics.

Programs are executed in the context of a *memory*, made of a variable store and a signal environment, and relative to a type environment. Formally, 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, where the first three components are defined next. Since Γ is meant to specify the types of variables and signals, which are formally introduced only in Section 3.1, let us just mention here that these types have the form δ *var* and δ *sig* respectively.

A *type environment* Γ is a mapping from a *finite* subset of names to the appropriate types. Since the set of names is infinite, it is always possible to find a fresh name not in the domain of Γ. We denote the update of Γ by a variable x of type δ *var* by $\{x : \delta \ var\}\Gamma$. Similarly we denote the update of Γ by a signal a of type δ *sig* by $\{a : \delta \ sig\}\Gamma$.

A *variable store* S is a mapping from a *finite* subset of variables to values, elements of a set Val. By abuse of language we denote by $S(e)$ the atomic evaluation of the expression e under S, which always terminates by definition of Exp. 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 belongs or not to the domain of S.

A *signal environment* E is a *finite* subset of signals, representing all signals that have been emitted during a given instant. A signal environment E may be updated in two ways: by including signal a into E when the statement `emit` a is executed, or by resetting E to $\emptyset$ when a new instant starts, as will be explained in Section 2.2.2. Note that E is not a multiset, so multiple emissions of the same signal during an instant are equivalent to a single emission.

Finally, a *memory* M is a pair $\langle S, E \rangle$, where S is a variable store and E is a signal environment.

We shall restrict our attention to well-formed configurations, defined next.

Definition 2.2 (Well-formed configuration)
A configuration $C = \langle \Gamma, S, E, P \rangle$ *is* well-formed *if it satisfies the conditions:*

i) $\mathtt{fs}(P) \subseteq \mathtt{dom}(\Gamma)$*;*

ii) $\mathtt{fv}(P) \subseteq \mathtt{dom}(S)$*;*

iii) $\mathtt{dom}(S) \cup E \subseteq \mathtt{dom}(\Gamma)$*.*

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* $\ddagger$ is defined inductively on pairs $\langle E, P\rangle$ by the rules in Figure 1. The meaning of $\langle E, P\rangle\ddagger$ is that program P is suspended under the signal environment E. Suspension is introduced by the construct ($\texttt{when}\ a\ \texttt{do}\ P$), in case signal a is not present in the signal environment. The suspension of a program P is propagated to the contexts ($\texttt{when}\ a\ \texttt{do}\ P$), ($\texttt{do}\ P\ \texttt{watching}\ a$), $(P\,;\,Q)$ and $(P \triangleleft Q)$. Processes of these forms will then be called *suspendable processes*.

We extend suspension to configurations by letting $\langle \Gamma, S, E, P\rangle\ddagger$ if $\langle E, P\rangle\ddagger$.

$$\text{(WHEN-SUS}_1\text{)}\ \frac{a \notin E}{\langle E, \texttt{when}\ a\ \texttt{do}\ P\rangle\ddagger} \qquad \text{(WHEN-SUS}_2\text{)}\ \frac{\langle E, P\rangle\ddagger}{\langle E, \texttt{when}\ a\ \texttt{do}\ P\rangle\ddagger}$$

$$\text{(WATCH-SUS)}\ \frac{\langle E, P\rangle\ddagger}{\langle E, \texttt{do}\ P\ \texttt{watching}\ a\rangle\ddagger}$$

$$\text{(SEQ-SUS)}\ \frac{\langle E, P\rangle\ddagger}{\langle E, P\,;\,Q\rangle\ddagger} \qquad \text{(PAR-SUS)}\ \frac{\langle E, P\rangle\ddagger \quad \langle E, Q\rangle\ddagger}{\langle E, P \triangleleft Q\rangle\ddagger}$$

Fig. 1. Suspension predicate

An *instant* is a sequence of moves leading all threads to termination or suspension. There are two forms of transitions between configurations: simple *moves*, denoted by the arrow $C \to C'$, and *instant changes*, denoted by $C \hookrightarrow C'$. These are collectively referred to as *steps* and denoted by $C \longmapsto C'$. The reflexive and transitive closures of these transition relations will be denoted with a '$*$' as usual. The next section defines simple moves and the following one defines instant changes.

2.2.1 Moves

The operational rules for deriving simple moves $C \to C'$ are given in Figures 2 and 3, for imperative and reactive constructs respectively. These rules describe the execution of programs within a given instant. 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`. The local variable declaration is similar to the local signal declaration, described next.

$$\text{(ASSIGN-OP)}\quad \langle \Gamma, S, E, x := e\rangle \to \langle \Gamma, \{x \mapsto S(e)\}S, E, \texttt{nil}\rangle$$

$$\text{(SEQ-OP}_1\text{)}\quad \langle \Gamma, S, E, \texttt{nil}\,;\, Q\rangle \to \langle \Gamma, S, E, Q\rangle$$

$$\text{(SEQ-OP}_2\text{)}\quad \frac{\langle \Gamma, S, E, P\rangle \to \langle \Gamma', S', E', P'\rangle}{\langle \Gamma, S, E, P\,;\, Q\rangle \to \langle \Gamma', S', E', P'\,;\, Q\rangle}$$

$$\text{(LET-OP)}\quad \frac{x' \notin \texttt{dom}(\Gamma)}{\langle \Gamma, S, E, \texttt{let } x : \delta = e \texttt{ in } P\rangle \to \langle \{x' \colon \delta\ var\}\Gamma, \{x' \mapsto S(e)\}S, E, \{x'/x\}P\rangle}$$

$$\text{(COND-OP}_1\text{)}\quad \frac{S(e) = \texttt{true}}{\langle \Gamma, S, E, \texttt{if } e \texttt{ then } P \texttt{ else } Q\rangle \to \langle \Gamma, S, E, P\rangle}$$

$$\text{(COND-OP}_2\text{)}\quad \frac{S(e) = \texttt{false}}{\langle \Gamma, S, E, \texttt{if } e \texttt{ then } P \texttt{ else } Q\rangle \to \langle \Gamma, S, E, Q\rangle}$$

$$\text{(WHILE-OP}_1\text{)}\quad \frac{S(e) = \texttt{true}}{\langle \Gamma, S, E, \texttt{while } e \texttt{ do } P\rangle \to \langle \Gamma, S, E, P\,;\, \texttt{while } e \texttt{ do } P\rangle}$$

$$\text{(WHILE-OP}_2\text{)}\quad \frac{S(e) = \texttt{false}}{\langle \Gamma, S, E, \texttt{while } e \texttt{ do } P\rangle \to \langle \Gamma, S, E, \texttt{nil}\rangle}$$

Fig. 2. Operational semantics of imperative constructs

Consider now the rules for the reactive constructs. Signal emission adds the emitted signal to the signal environment: this signal will then be viewed as present until the end of the instant. The local signal declaration adds to the type environment a fresh signal name, with the declared type. Note that this name is *not* added to the signal environment: this means that the signal is *known* (and thus not available for a further declaration), but it is not considered as present as long as it has not been emitted. The way we handle local names may seem non standard here: in fact, it mimics what is done in most implementations and allows for a greater accuracy in the definition of noninterference, as we shall see in Section 3.

The `watching` construct allows the execution of its body until this terminates or an instant change occurs; in the latter case its execution will abort or

$$\text{(EMIT-OP)}\quad \langle \Gamma, S, E, \texttt{emit}\ a\rangle \to \langle \Gamma, S, \{a\} \cup E, \texttt{nil}\rangle$$

$$\text{(LOCAL-OP)}\quad \frac{a' \notin \texttt{dom}(\Gamma)}{\langle \Gamma, S, E, \texttt{local}\ a : \delta\ \texttt{in}\ P\rangle \to \langle \{a' : \delta\ sig\}\Gamma, S, E, \{a'/a\}P\rangle}$$

$$\text{(WATCH-OP}_1\text{)}\quad \langle \Gamma, S, E, \texttt{do nil watching}\ a\rangle \to \langle \Gamma, S, E, \texttt{nil}\rangle$$

$$\text{(WATCH-OP}_2\text{)}\quad \frac{\langle \Gamma, S, E, P\rangle \to \langle \Gamma', S', E', P'\rangle}{\langle \Gamma, S, E, \texttt{do}\ P\ \texttt{watching}\ a\rangle \to \langle \Gamma', S', E', \texttt{do}\ P'\ \texttt{watching}\ a\rangle}$$

$$\text{(WHEN-OP}_1\text{)}\quad \frac{a \in E}{\langle \Gamma, S, E, \texttt{when}\ a\ \texttt{do nil}\rangle \to \langle \Gamma, S, E, \texttt{nil}\rangle}$$

$$\text{(WHEN-OP}_2\text{)}\quad \frac{a \in E \quad \langle \Gamma, S, E, P\rangle \to \langle \Gamma', S', E', P'\rangle}{\langle \Gamma, S, E, \texttt{when}\ a\ \texttt{do}\ P\rangle \to \langle \Gamma', S', E', \texttt{when}\ a\ \texttt{do}\ P'\rangle}$$

$$\text{(PAR-OP}_1\text{)}\quad \langle \Gamma, S, E, \texttt{nil} \leftthreetimes Q\rangle \to \langle \Gamma, S, E, Q\rangle$$

$$\text{(PAR-OP}_2\text{)}\quad \frac{\langle \Gamma, S, E, P\rangle \to \langle \Gamma', S', E', P'\rangle}{\langle \Gamma, S, E, P \leftthreetimes Q\rangle \to \langle \Gamma', S', E', P' \leftthreetimes Q\rangle}$$

$$\text{(PAR-OP}_3\text{)}\quad \frac{\langle E, P\rangle\ddagger \quad \neg\langle E, Q\rangle\ddagger}{\langle \Gamma, S, E, P \leftthreetimes Q\rangle \to \langle \Gamma, S, E, Q \leftthreetimes P\rangle}$$

Fig. 3. Operational semantics of reactive constructs

resume at the next instant depending on the presence or not of the tested signal (as will be explained in Section 2.2.2). The `when` construct executes its body under the control of a signal: if the signal is present it proceeds, otherwise it suspends. Note that both the `when` and `watching` operators are *static*, in the sense that they remain present after a transition if their body is different from `nil`. However, while the `when` construct checks the presence of its signal at every step, the `watching` construct only checks it at the end of the instant (this is the reason why the test does not appear in the rules of Figure 3 but rather in those of Figure 4).

The synchronous parallel composition $\leftthreetimes$ implements a kind of *co-routine mechanism*. It executes its left component until termination or suspension, and then gives the control to its right component, provided this one is not already sus-

pended (this can happen for instance if the right component is of the form (`when` a `do` R) and signal a is absent). Technically, the passage of control is achieved by swapping the two components. Since each component can emit signals that unblock the other, the control keeps alternating between the two components until neither of them can move any more. At this point the whole program is either terminated or suspended. As noted in the Introduction, the operator $\curvearrowleft$ incorporates a *cooperative scheduling* discipline. To work fairly, it requires each thread to "play the game" and yield the control after a finite number of steps : indeed, in the program $P \curvearrowleft Q$, if P does not terminate then Q will never get the control. In recent studies, conditions have been proposed to ensure both a fair and tractable behaviour of reactive threads [8,7]. We shall leave this question aside here, and we will not make any assumption about the behaviour of threads.

Let us illustrate the use of reactive constructs with a couple of examples. Here we focus on the operational semantics of programs and thus we shall not be concerned with security levels, hence we do not consider the type environment. Similarly, in these two examples the variable store is not relevant, thus we ignore it, while the signal environment is assumed to be initially empty.

The first example emphasizes the alternation of control that results from the combination of the $\curvearrowleft$ and `when` constructs, and the second shows that the operator $\curvearrowleft$ is not associative, that is, that the grouping of components matters. We shall use underbraces to indicate the suspension of the executing thread (the one in head position).

Example 1 (Thread alternation) *In this example three threads are queuing for execution in an empty signal environment (left column). The first two threads are suspended and thus so is their composition. Then by rule* (PAR-OP$_3$) *the third thread pops in head position and emits signal a reducing to* `nil`*, and then disappearing (rules* (EMIT-OP)*,* (PAR-OP$_2$) *and* (PAR-OP$_1$)*). Now the remaining two threads can execute one after the other (we let the reader work out the rules to be used) until termination of the whole program. Summing up, the emission of signal a by the third thread unblocks the first suspended thread, which in turn unblocks the second.*

	$\{\}$	$(\underbrace{(\texttt{when}\ a\ \texttt{do emit}\ b) \curvearrowleft (\texttt{when}\ b\ \texttt{do emit}\ c)}) \curvearrowleft \texttt{emit}\ a$
$\rightarrow$	$\{\}$	$\texttt{emit}\ a \curvearrowleft ((\texttt{when}\ a\ \texttt{do emit}\ b) \curvearrowleft (\texttt{when}\ b\ \texttt{do emit}\ c))$
$\rightarrow^\star$	$\{a\}$	$(\texttt{when}\ a\ \texttt{do emit}\ b) \curvearrowleft (\texttt{when}\ b\ \texttt{do emit}\ c)$
$\rightarrow^\star$	$\{a, b\}$	$\texttt{when}\ b\ \texttt{do emit}\ c$
$\rightarrow^\star$	$\{a, b, c\}$	$\texttt{nil}$

Example 2 (Synchronous composition $\curlywedge$ is not associative) *The operator $\curlywedge$ is non commutative with respect to execution traces, as shown by the programs* (`emit` a $\curlywedge$ `emit` b) *and* (`emit` b $\curlywedge$ `emit` a)*, which emit signals a and b in different orders. It is also non associative, as shown by the programs:*

((`when` a `do emit` b) $\curlywedge$ `emit` a) $\curlywedge$ `emit` c (`when` a `do emit` b) $\curlywedge$ (`emit` a $\curlywedge$ `emit` c)

The first program emits the signals in the order a, b, c, while the second emits them in the order a, c, b. Here the operational rules that come into play are the three rules for the operator $\curlywedge$, the rules (WHEN-OP$_1$) *and* (EMIT-OP)*, as well as the clause* (WHEN-SUS$_1$) *for the suspension predicate.*

We have seen that suspension of a thread may be lifted during an instant upon emission of the required 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, the program (or rather, the configuration) is said to be *inactive* and 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 (`do` P `watching` a) whose tested signal a is present are killed, and all signals are reset to absent, that is, the new signal environment becomes the empty set. The `watching` construct provides a mechanism to recover from suspension and from deadlock situations originated by `when` commands, as will be illustrated by Example 4 below.

The semantics of *instant changes* is described in Figure 4. The operational rule (INSTANT-OP) is the only rule by which a transition $C \hookrightarrow C'$ can be derived. It specifies how a suspended configuration can evolve, in the transition from an instant to the next, to a new configuration, possibly active since the function $\lfloor P \rfloor_E$ may prune off some suspended parts of P. The function $\lfloor P \rfloor_E$ is meant to be applied to suspended processes (see Figure 1) and therefore it is defined only for the subset of programs that can suspend. It may easily be seen, by inspection of the various clauses, that if P contains no `watching` commands, then $\lfloor P \rfloor_E$ coincides with P. Note that the two clauses for the statement (`when` a `do` P) correspond to the two reasons why this program may suspend: in particular, if the reason is the absence of signal a, then the control is blocked on this waiting point and thus the body P cannot contain suspended subprograms. This point is best illustrated with an example:

Example 3 *Let $E = \{b, c\}$ and P be the program defined as follows:*

$$P = \texttt{do}\ (\texttt{when}\ b\ \texttt{do}\ (\texttt{do}\ (\texttt{when}\ d\ \texttt{do}\ Q)\ \texttt{watching}\ c))\ \texttt{watching}\ a$$

We let the reader verify that $\lfloor P \rfloor_E =$ `do` (`when` b `do nil`) `watching` a.

$$\text{(INSTANT-OP)}\ \frac{\langle E, P\rangle\ddagger}{\langle \Gamma, S, E, P\rangle \hookrightarrow \langle \Gamma, S, \emptyset, \lfloor P \rfloor_E\rangle}\ \text{where}$$

$$\lfloor \texttt{do}\ P\ \texttt{watching}\ a \rfloor_E \stackrel{\text{def}}{=} \begin{cases} \texttt{nil} & \text{if } a \in E \\ \texttt{do}\ \lfloor P \rfloor_E\ \texttt{watching}\ a & \text{otherwise} \end{cases}$$

$$\lfloor \texttt{when}\ a\ \texttt{do}\ P \rfloor_E \stackrel{\text{def}}{=} \begin{cases} \texttt{when}\ a\ \texttt{do}\ \lfloor P \rfloor_E & \text{if } a \in E \\ \texttt{when}\ a\ \texttt{do}\ P & \text{otherwise} \end{cases}$$

$$\lfloor P\,;\, Q \rfloor_E \stackrel{\text{def}}{=} \lfloor P \rfloor_E\,;\, Q$$

$$\lfloor P \leftcurvearrow Q \rfloor_E \stackrel{\text{def}}{=} \lfloor P \rfloor_E \leftcurvearrow \lfloor Q \rfloor_E$$

Fig. 4. Operational semantics of instant changes

In the sequel we assume programs run in the empty signal environment if not otherwise specified. The following is an example where an instant change breaks a causality cycle:

Example 4 (*Break of causality cycle*) *Consider the following program, where a causality cycle is initially present between the emissions of signals b and c:*

$$\texttt{emit}\ a\,;\, ((\texttt{when}\ b\ \texttt{do emit}\ c) \leftcurvearrow (\texttt{do}\ (\texttt{when}\ c\ \texttt{do emit}\ b)\ \texttt{watching}\ a)\,;\, \texttt{emit}\ b)$$

Here the whole program suspends after the emission of a. Then, since a is present, 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 an instant thanks to the `watching` construct.

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

Example 5 (*The primitive* `pause`) *The program* `pause` *is defined by:*

$$\texttt{pause} = \texttt{local}\ a : \delta\ \texttt{in}\ (\texttt{local}\ b : \theta\ \texttt{in}\ (\texttt{emit}\ b\,;\ \texttt{do}\ (\texttt{when}\ a\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b))$$

Here the local declaration of signal a ensures that the signal cannot be emitted outside the scope of its declaration, and thus the program will suspend when reaching the subprogram (`when` a `do nil`)*. At this point, the presence of b is checked: since it has been emitted, the subprogram* (`when` a `do nil`) *is aborted at the beginning of the next instant. Formally, the execution of* `pause` *goes as follows:*

$$\begin{array}{rl|l}
 & \{\} & \texttt{local}\ a : \delta\ \texttt{in}\ (\texttt{local}\ b : \theta\ \texttt{in}\ (\texttt{emit}\ b\,;\ \texttt{do}\ (\texttt{when}\ a\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b)) \\
\rightarrow^\star & \{\} & \texttt{emit}\ b\,;\ \texttt{do}\ (\texttt{when}\ a\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b \\
\rightarrow^\star & \{b\} & \underbrace{\texttt{do}\ (\texttt{when}\ a\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b} \\
\hookrightarrow & \{\} & \texttt{nil}
\end{array}$$

We may note here that the program `pause` does not suspend immediately but only after performing a few "administrative moves". This implies for instance that the program (`pause`; P) $\triangleleft$ (`pause`; Q) evolves to the program $Q \triangleleft P$ after a change of instant, since the second component gets the control before the suspension of the whole program. On the other hand, no switch of control occurs in (`pause`; P) $\triangleleft$ (`when` a `do` Q) since in this case the second component suspends immediately. Hence this program will evolve to $P \triangleleft$ (`when` a `do` Q) at instant change.

To see this style of programming at work, let us now consider a more practical example, adapted from [16]. Suppose we want to model the *replacement of a service* during execution (also called "hot-plug replacement"). We assume the service, henceforth called standard service, to run until the emission of a signal *switch*, which can be viewed as representing an internal failure or an external interrupt. At this point, a supply service takes over and runs for a fixed amount of time. We want the replacement to preserve two properties: (1) *continuity of service*, meaning that there is no temporal delay between the end of the first service and the start of the second, and (2) *coherence*, meaning that there is no overlapping between the two services.

Example 6 (*A practical example: service replacement)*

We model a service replacement system. Let $Service_1$ *and* $Service_2$ *denote respectively the standard service and the supply service. We consider an elementary notion of service, which consists simply in incrementing a counter. This is admittedly oversimplified, since we abstract even from the fact that a service responds to a client's request. In practice, this simple instruction will be replaced by a more elaborate finite task, triggered by an external request. Each* $Service_i$ *follows a basic protocol: it waits for signal* go_i *to be emitted and then, at every instant, it increments the variable* $count_i$ *and suspends until the end of the instant. As long as* go_i *is not present (a period that can span through several instants),* $Service_i$ *is blocked. However, the services are not completely symmetric:* $Service_1$ *can be killed, at the end of the instant where signal kill is emitted, while* $Service_2$ *terminates on its own, in a delay* $d \geq 1$ *after the emission of* go_2*. Formally, the services are defined by:*

$$Service_1 = \texttt{do}\,(\texttt{await}\ go_1\,;\texttt{loop}\ (count_1 := count_1 + 1\,;\ \texttt{pause}))\ \texttt{watching}\ kill$$

$$Service_2 = \ \texttt{await}\ go_2\,;\ j := 0\,;$$
$$\texttt{while}\ j < d\ \texttt{do}\ (j := j + 1\,;\ count_2 := count_2 + 1\,;\ \texttt{pause})$$

The system includes two more components: Service-Switcher, which emits the signal switch at a predefined time $t \geq 1$*, and Service-Controller, which triggers* $Service_1$ *via signal* go_1*, and then stops it via signal kill when switch is emitted, nsubsequently triggering* $Service_2$ *via signal* go_2*. Formally:*

$$\textit{Service-Switcher} = k := 1\,;\ \texttt{while}\ k < t\ \texttt{do}\ (k := k + 1;\ \texttt{pause})\,;\ \texttt{emit}\ switch$$

$$\textit{Service-Controller} = \ \texttt{emit}\ go_1\,;\ \texttt{await}\ switch;\ \texttt{emit}\ kill;\ \texttt{pause}\,;\ \texttt{emit}\ go_2$$

The whole system is then defined by:

$$\textit{SR-System} = (\textit{Service-Switcher} \curlywedge \textit{Service-Controller}) \curlywedge (Service_1 \curlywedge Service_2)$$

Note that both services are programmed to increment their counter at least once. Signal kill is emitted at the t*-th instant of the system's execution, while* go_2 *is emitted at the following instant, due to the intervening* `pause` *instruction. The reason for postponing the emission of* go_2 *is that, since* $Service_1$ *is killed only at the end of instant* t*, the start of* $Service_2$ *should be delayed till instant* $t + 1$*, otherwise the property of coherence would not hold. Note also that both Service-Switcher and Service-Controller terminate as soon as their job is completed, that is, at time* t *and* $t + 1$ *respectively.*

It is now easy to convince oneself that *SR-System* satisfies the properties of continuity of service and coherence, since $Service_1$ stops at the end of instant t and $Service_2$ starts at the beginning of instant $t+1$. Indeed, the synchronous computational model ensures that $Service_1$ actually stops at time t, that is, at the same time when the *switch* signal is emitted, as witnessed by the final value of the counter $count_1$. Such a behaviour would not be possible to program in an asynchronous model, since in that case $Service_1$ could always "refuse" to receive signal *kill* even though it has been emitted.

2.2.3 Computations

In this section, we establish a few properties of computations. An important property of reactive computations is their *determinacy* (up to a renaming of local variables). Indeed, this property is often put forward as an advantage of synchronous programming over other styles of concurrent programming. We start by giving some definitions.

Definition 2.3 (States of a configuration)
A configuration C is said to be:

- active*, if there exists C' such that $C \to C'$, which we denote by $C \to$;*
- alive*, if there exists C' such that $C \longmapsto C'$, which we denote by $C \longmapsto$;*
- terminated*, if it is not alive, which we denote by $C \mapsto\!\!\!\!\!\!/\;$.*

Similarly, we denote by $C \hookrightarrow$ the fact that there exists C' such that $C \hookrightarrow C'$. Clearly, if C is active then it is also alive. If a configuration $C = \langle \Gamma, S, E, P \rangle$ is able to perform a step, the form of this step, simple move or instant change, will depend on whether P is suspended or not in the signal environment E. This leads us to the following definition:

Definition 2.4 (Computation)
A computation with initial configuration $C = \langle \Gamma, S, E, P \rangle$ is a sequence of transitions of the form:

$$\langle \Gamma, S, E, P \rangle \to^\star \langle \Gamma_1, S_1, E_1, P_1 \rangle \hookrightarrow \langle \Gamma_1, S_1, \emptyset, \lfloor P_1 \rfloor_{E_1} \rangle \to^\star \langle \Gamma_2, S_2, E_2, P_2 \rangle \hookrightarrow \ldots$$

where all the configurations $\langle \Gamma_i, S_i, E_i, P_i \rangle$ are suspended.

We establish now a few properties of computations. Recall that a memory is a pair $\langle S, E \rangle$. It is easy to see from the semantic rules that computations may affect a type environment only by extending it with a fresh name. Similarly, they may affect a store only by updating it (changing the value of a variable or extending it with a fresh variable). As concerns the signal environment, this may only be affected by the addition of a new signal, in case of simple moves,

or by a reset to $\emptyset$, in case of an instant change. These facts are summarized in the following proposition, which we state without proof. By abuse of notation, we use $\{n : \delta\ name\}\Gamma$ to stand for either $\{x : \delta\ var\}\Gamma$ or $\{a : \delta\ sig\}\Gamma$.

Proposition 2.1 (Simple properties of computations)

(1) If $\langle \Gamma, S, E, P\rangle \longmapsto \langle \Gamma', S', E', P'\rangle$, *then* $\Gamma' = \Gamma$ *or* $\Gamma' = \{n : \delta\ name\}\Gamma$ *for some* $n \notin \mathtt{dom}(\Gamma)$.

(2) If $\langle \Gamma, S, E, P\rangle \longmapsto \langle \Gamma', S', E', P'\rangle$, *then* $\mathtt{dom}(S') = \mathtt{dom}(S)$ *or there exists* $x \notin \mathtt{dom}(S)$ *such that* $\mathtt{dom}(S') = \mathtt{dom}(S) \cup \{x\}$.

(3) If $\langle \Gamma, S, E, P\rangle \rightarrow \langle \Gamma', S', E', P'\rangle$, *then* $E' = E$ *or* $E' = E \cup \{a\}$ *for some* $a \in \mathtt{dom}(\Gamma)\backslash E$.

We show that computations preserve well-formedness of configurations. Recall from Definition 2.2 that $C = \langle \Gamma, S, E, P\rangle$ is well-formed if $\mathtt{fs}(P) \subseteq \mathtt{dom}(\Gamma)$, $\mathtt{fv}(P) \subseteq \mathtt{dom}(S)$ and $\mathtt{dom}(S) \cup E \subseteq \mathtt{dom}(\Gamma)$.

Proposition 2.2 (Well-formedness is preserved by computations)

If C *is a well-formed configuration and* $C \longmapsto C'$ *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 $\mathtt{fs}(P') \subseteq \mathtt{dom}(\Gamma')$, $\mathtt{fv}(P') \subseteq \mathtt{dom}(S')$ and $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$. 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 $\mathtt{fv}(\lfloor P \rfloor_E) \subseteq \mathtt{fv}(P)$. Then the required 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-OP$_2$), (WATCH-OP$_2$), (WHEN-OP$_2$) and (PAR-OP$_2$). Note that in all base cases apart from (LET-OP), we have $\mathtt{fv}(P') \subseteq \mathtt{fv}(P)$ and $\mathtt{dom}(S') = \mathtt{dom}(S)$, so the property $\mathtt{fv}(P') \subseteq \mathtt{dom}(S')$ is trivial. Similarly, in all base cases apart from (LET-OP) and (LOCAL-OP), we have $\mathtt{fs}(P') \subseteq \mathtt{fs}(P)$ and $\mathtt{dom}(\Gamma') = \mathtt{dom}(\Gamma)$, so the property $\mathtt{fs}(P') \subseteq \mathtt{dom}(\Gamma')$ is trivial. As for the property $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$, it is trivial in all cases where $\Gamma' = \Gamma, S' = S$ and $E' = E$. We examine some of the remaining cases.
   - (ASSIGN-OP) Here $\Gamma' = \Gamma, \mathtt{dom}(S') = \mathtt{dom}(S)$ and $E' = E$, hence the property $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$ follows immediately from that for C.
   - (SEQ-OP$_2$) 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 $\mathtt{fs}(P_1') \subseteq \mathtt{dom}(\Gamma')$, $\mathtt{fv}(P_1') \subseteq \mathtt{dom}(S')$ and $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$. Hence the

last property for C' is already given. Now, since C is well-formed, we know that $\mathtt{fs}(P_2) \subseteq \mathtt{dom}(\Gamma)$ and $\mathtt{fv}(P_2) \subseteq \mathtt{dom}(S)$. By Proposition 2.1 $\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma')$ and $\mathtt{dom}(S) \subseteq \mathtt{dom}(S')$, whence $\mathtt{fs}(P_1'; P_2) = \mathtt{fs}(P_1') \cup \mathtt{fs}(P_2') \subseteq \mathtt{dom}(\Gamma')$ and $\mathtt{fv}(P_1'; P_2) = \mathtt{fv}(P_1') \cup \mathtt{fv}(P_2') \subseteq \mathtt{dom}(S')$.

- (LET-OP) Here $P = \mathtt{let}\ x : \delta = e\ \mathtt{in}\ P_1$ and $P' = \{x'/x\}P_1$, for some x' not in $\mathtt{dom}(\Gamma)$ and thus not in $\mathtt{dom}(S)$ nor in $\mathtt{fv}(P)$. We have $\Gamma' = \{x' : \delta\ var\}\Gamma$, $S' = \{x' \mapsto S(e)\}S$, $E' = E$. Since $\mathtt{fs}(P') = \mathtt{fs}(P)$ and $\mathtt{fv}(P') = \mathtt{fv}(P) \cup \{x'\}$, it follows that $\mathtt{fs}(P') \subseteq \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma')$ and $\mathtt{fv}(P') \subseteq \mathtt{dom}(S) \cup \{x'\} = \mathtt{dom}(S')$. Similarly, we have $\mathtt{dom}(S') \cup E' = \mathtt{dom}(S) \cup \{x'\} \cup E \subseteq \mathtt{dom}(\Gamma) \cup \{x'\} = \mathtt{dom}(\Gamma')$.
- (EMIT-OP) Here $P = \mathtt{emit}\ a$ and we have $\Gamma' = \Gamma, S' = S$ and $E' = E \cup \{a\}$. Since C is well-formed, we know that $\mathtt{fs}(P) \subseteq \mathtt{dom}(\Gamma)$, hence $a \in \mathtt{dom}(\Gamma)$. We can then conclude that $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$.
- (LOCAL-OP) Here $P = \mathtt{local}\ a : \delta\ \mathtt{in}\ P_1$ and $P' = \{a'/a\}P_1$ for some a' not in $\mathtt{dom}(\Gamma)$. Since $\Gamma' = \{a' : \delta\ sig\}\Gamma$, $S' = S$ and $E' = E$, we have $\mathtt{fs}(P') = \mathtt{fs}(P) \cup \{a'\} \subseteq \mathtt{dom}(\Gamma) \cup \{a'\} = \mathtt{dom}(\Gamma')$ and $\mathtt{dom}(S') \cup E' = \mathtt{dom}(S) \cup E \subseteq \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma')$.
- (WATCH-OP$_2$) Here $P = \mathtt{do}\ P_1\ \mathtt{watching}\ a$, $P' = \mathtt{do}\ P_1'\ \mathtt{watching}\ a$ and $C \to C'$ is deduced from $\langle \Gamma, S, E, P_1 \rangle \to \langle \Gamma', S', E', P_1' \rangle$. By induction $\mathtt{fs}(P_1') \subseteq \mathtt{dom}(\Gamma')$, $\mathtt{fv}(P_1') \subseteq \mathtt{dom}(S')$ and $\mathtt{dom}(S') \cup E' \subseteq \mathtt{dom}(\Gamma')$. Thus the third property is given. Moreover $\mathtt{fv}(P') = \mathtt{fv}(P_1')$, so we only have to prove the first property. By Proposition 2.1 $\mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma')$. Since C is well-formed $\mathtt{fs}(P) \subseteq \mathtt{dom}(\Gamma)$, thus $a \in \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Gamma')$. Whence $\mathtt{fs}(P') = \mathtt{fs}(P_1') \cup \{a\} \subseteq \mathtt{dom}(\Gamma') \cup \{a\} = \mathtt{dom}(\Gamma')$.

□

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`.

Proposition 2.3 *A configuration $C = \langle \Gamma, S, E, P \rangle$ is* terminated *if and only if $P = \mathtt{nil}$.*

Proof We prove that $P \neq \mathtt{nil}$ implies $C \longmapsto$, by induction on the structure of P. Recall that by definition $C = \langle \Gamma, S, E, P \rangle \hookrightarrow$ if and only if $\langle E, P \rangle \ddagger$. We examine some sample cases.

- $P = x := e$. Since C is well-formed, $\mathtt{fv}(e) \subseteq \mathtt{dom}(S)$ and thus the value $S(e)$ is defined. Then $C \to \langle \Gamma, \{x \mapsto S(e)\}S, E, \mathtt{nil} \rangle$ by (ASSIGN-OP).
- $P = P_1\,;\ P_2$. If $P_1 = \mathtt{nil}$, then $C \to \langle \Gamma, S, E, P_2 \rangle$ by (SEQ-OP$_1$). If $P_1 \neq \mathtt{nil}$ then by induction either $(E, P_1)\ddagger$ or there exist Γ', S', E', P_1' such that $\langle \Gamma, S, E, P_1 \rangle \to \langle \Gamma', S', E', P_1' \rangle$. In the first case we have $(E, P_1; P_2)\ddagger$ by

(SEQ-SUS) and $C \hookrightarrow \langle \Gamma, S, \emptyset, \lfloor P_1 \rfloor_E ; P_2 \rangle$ by (INSTANT-OP), while in the latter we have $C \rightarrow \langle \Gamma', S', E', P_1' ; P_2 \rangle$ by (SEQ-OP$_2$).

- $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ P_1$. Since $\texttt{dom}(\Gamma)$ is finite and Var is infinite, we can always find x' such that $x' \notin \texttt{dom}(\Gamma)$, hence we may apply (LET-OP) to deduce $C \rightarrow$.
- $P = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. By well-formedness 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-OP$_1$) or rule (COND-OP$_2$) depending on the value $S(e)$.
- $P = \texttt{when}\ a\ \texttt{do}\ P_1$. If $a \notin E$, then we have $(E, \texttt{when}\ a\ \texttt{do}\ P_1)\ddagger$ by (WHEN-SUS) and by (INSTANT-OP) we deduce $C \hookrightarrow \langle \Gamma, S, \emptyset, \lfloor \texttt{when}\ a\ \texttt{do}\ P_1 \rfloor_E \rangle$. Assume now $a \in E$. If $P_1 = \texttt{nil}$, then $C \rightarrow \langle \Gamma, S, E, \texttt{nil} \rangle$ by (WHEN-OP$_1$). If $P_1 \neq \texttt{nil}$, then we use induction exactly as in the case $P = P_1 \,;\, P_2$.
- $P = P_1 \leftthreetimes P_2$. If $P_1 = \texttt{nil}$, then $C \rightarrow \langle \Gamma, S, E, P_2 \rangle$ by rule (PAR-OP$_1$). If $P_1 \neq \texttt{nil}$ then by induction either $(E, P_1)\ddagger$ 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 latter case $C \rightarrow \langle \Gamma', S', E', P_1' \leftthreetimes P_2 \rangle$ by (PAR-OP$_2$). In the former case either also $(E, P_2)\ddagger$ and thus $(E, P_1 \leftthreetimes P_2)\ddagger$ by (PAR-SUS) and $C \hookrightarrow \langle \Gamma, S, \emptyset, \lfloor P_1 \rfloor_E \leftthreetimes \lfloor P_2 \rfloor_E \rangle$ by (INSTANT-OP), or $\neg(E, P_2)\ddagger$ and $C \rightarrow \langle \Gamma, S, E, P_2 \leftthreetimes P_1 \rangle$ by (PAR-OP$_3$).

□

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

Proposition 2.4 (Determinism) *Let $C = \langle \Gamma, S, E, P \rangle$ be an alive configuration. Then:*

- *If C is* suspended*, then there exists a unique C' such that $C \hookrightarrow C'$;*
- *If C is* active*, then there exists a configuration $C' = \langle \Gamma', S', E', P' \rangle$ such that $C \rightarrow C'$, and for any $C'' = \langle \Gamma'', S'', E'', P'' \rangle$ such that $C \rightarrow C''$, one of the following holds:*
  - *$C'' = C'$;*
  - *for some $a, b \in Sig \backslash \texttt{dom}(\Gamma)$, $\texttt{dom}(\Gamma') \backslash \texttt{dom}(\Gamma) = \{a\}$, $\texttt{dom}(\Gamma'') \backslash \texttt{dom}(\Gamma) = \{b\}$ and $C'' = \{b/a\}C'$;*
  - *for some $x, y \in Var \backslash \texttt{dom}(\Gamma)$, $\texttt{dom}(\Gamma') \backslash \texttt{dom}(\Gamma) = \{x\}$, $\texttt{dom}(\Gamma'') \backslash \texttt{dom}(\Gamma) = \{y\}$, $C'' = \{y/x\}C'$ and $S''(y) = S'(x)$.*

Proof By Proposition 2.3 we know that $P \neq \texttt{nil}$. If $\langle E, P\rangle\ddagger$, 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 $\neg\langle E, P\rangle\ddagger$, 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 present our security type system and prove some of its properties. We then formalise our noninterference property as a form of self-bisimilarity, and prove that the type system guarantees this property. Finally, we compare our notion of security with a more standard one, and show that our property is stronger (and thus closer to the notion of typability) in several respects.

3.1 Type System

The role of a security type system is to rule out insecure programs. Now, what exactly should be the security property in our language? As usual, we shall require that a *low observer*, i.e. a potentially malicious one, should only have access to the low memory, that is, in our setting, to the state of low variables and low signals. It remains to establish at which points of execution this access could take place. Since reactive computations are described by a small-step semantics and involve two kinds of transitions, within instants and across instants, several possibilities can be envisaged.

For instance, observation could be restricted to *final states* of computations, as is the case for sequential languages [27], or, on the opposite, it could be taken to cover all *intermediate states* of possibly nonterminating computations, as is the case for most concurrent languages [24,22,14]). The choice is not immediately obvious, since synchronous concurrency in our language amounts to a deterministic interleaving of cooperative threads, and thus some of the well known problems of input-output semantics in a concurrency scenario (like the non-reproducibility of results and the lack of compositionality) do not arise in the reactive setting. However, it is clear that plain input-output semantics is not appropriate for reactive programming, for the same reason for which it is not suited for other kinds of concurrent programming: indeed, in a

concurrent setting 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 along a sequence of 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[3]. In other words, a "good" nonterminating reactive program should span over an infinity of instants. One could then envisage adopting an intermediate semantics, where states are observed only at the end of instants (during which computations are guaranteed to terminate if programs are well-behaved), over a possibly infinite sequence of instants. We shall leave this question open for future investigation, and adopt here a *fine-grained observation* where all states are taken into account, as in previous work on concurrent languages. Note that a compositional static analysis will itself verify a fine-grained security property, since each instruction is individually checked.

We may then informally define *noninterference* to be the property of programs whose low memory (value of low variables and presence of low signals) is never affected, at any stage of execution, by changes in the high memory (value of high variables or presence of high signals). With this intuition in mind, let us now proceed to identify the kinds of insecure information flows, or information *leaks*, which may arise in reactive programs. We shall essentially consider here succinct examples, grouping them into categories for easy reference. For more interesting examples the reader may want to look back at those presented in the Introduction or in Section 2. Later in this section, we shall reexamine the service-replacement example of Section 2, and establish under which assignments of security levels it may be considered secure.

In the next examples, the derived program `pause` will be used as if it were a primitive construct. This is justified by the fact that `pause` is indeed usually assumed to be primitive in reactive languages (we chose here to define it as a derived construct to emphasize the expressiveness of our language). Moreover, for any program context it is possible to type `pause` in such a way that it is compatible with that context, in the sense that the program obtained by plugging `pause` in the context is typable.

Explicit flow $\quad y_L := x_H$

Here the value of x_H is directly written into y_L. This kind of flow is also called *direct flow* and can be excluded by a very simple typing rule. Note that the introduction of signals does not generate new explicit flows. Indeed, there is

[3] This is because, as we saw earlier, such an instantly diverging thread would keep the control forever, thus preventing all the other threads from executing.

no direct way to equalize the status (present or absent) of two signals. This can only be done by means of the constructs `when` or `watching`, as we shall see next.

Simple implicit flows

(5.1) $\texttt{if}\ x_H = 0\ \texttt{then}\ y_L := 0\ \texttt{else}\ y_L := 1$

(5.2) $\texttt{while}\ x_H \neq 0\ \texttt{do}\ (\, y_L := y_L + 1\,;\, x_H := 0\,)$

} imperative constructs

(5.3) $(\texttt{when}\ a_H\ \texttt{do emit}\ b_L) \nwarrow (\texttt{emit}\ c_L\,;\ \texttt{emit}\ a_H)$

(5.4) $\texttt{emit}\ c_L\,;\, (\texttt{do}\ (\texttt{when}\ a_H\ \texttt{do emit}\ b_L)\ \texttt{watching}\ c_L)$

(5.5) $\texttt{do}\ (\texttt{when}\ b_L\ \texttt{do emit}\ c_L)\ \texttt{watching}\ a_H \nwarrow (\texttt{pause}\,;\ \texttt{emit}\ b_L)$

} reactive constructs

Implicit flows are induced by the control structure of a program. The first two programs are classical examples of implicit flows associated with conditionals and while loops. In both cases the final value of y_L depends on the initial value of x_H. Note that if a terminating program is found to be insecure by looking at its final states, it will a fortiori be insecure under a more fine-grained observation. The standard way for ruling out these kinds of flows in imperative programs consists in forbidding low assignments in the body of conditionals or loops with a high condition. We shall use a similar rule here, forbidding also low signal emissions in the body of such statements.

Let us now consider the last three programs, which only involve reactive constructs. Note that under a fine-grained observation the order in which signals are emitted is observable. Suppose we execute program (5.3) in the empty signal environment. Then, since a_H is initially absent, the first component suspends and the the second component emits c_L and then a_H, thus unblocking the first component which can now emit b_L. If instead we execute program (5.3) in the signal environment $\{a_H\}$, then b_L will be emitted before c_L. Hence program (5.3) is insecure, because so is its component $(\texttt{when}\ a_H\ \texttt{do emit}\ b_L)$.

Program (5.4) is a variant of Example (8) discussed in the Introduction. Here, if a_H is present then b_L is emitted and the program terminates at the first instant. If a_H is absent then the program suspends and the body of the `watching` construct is killed at the end of the instant. In this case the signal b_L is not emitted. Hence this program is insecure, again because of $(\texttt{when}\ a_H\ \texttt{do emit}\ b_L)$.

It may be worth noting here that our fine-grained observation is *insensitive to the program status*, i.e. unable to detect whether a program is terminated, suspended or active in a given memory state. Under such an observation, the program (`when` a_H `do emit` b_L) is insecure even when run in isolation. In fact, if a_H is present then the program moves and emits b_L, whereas if a_H is absent it stays in its initial state and this state is observed (as producing no low change) even though it is suspended. However, we chose here to use terminating programs to illustrate implicit flows, as these may be easier to comprehend for the reader at this stage. Moreover it is interesting to have examples of programs that are insecure for any kind of observation.

Consider now program (5.5). If we run it in the empty signal environment, then the first component suspends and the second takes over, suspending after a few steps. At this point an instant change occurs and since a_H is absent, the first component remains unchanged. On the other hand the second component can now move and emit b_L, unblocking the first component which can then emit c_L and terminate. Suppose now program (5.5) runs in the signal environment $\{a_H\}$. In this case, when the instant change occurs, the first component reduces to `nil` and thus only signal b_L is emitted in the whole computation. Hence program (5.5) is insecure, because of its `watching` subprogram.

To rule out implicit flows in `when` and `watching` statements, we shall use the same rule as for conditionals and loops, namely forbid low signal emissions and low assignments in the body of `when` and `watching` statements whose controlling signal is high.

Note that all the above implicit flows are due to the flow of control within a single command, which tests a high variable or signal and performs a low memory change in its body. We turn now to more complex indirect flows, associated with nontermination, suspension and scheduling.

Nontermination leaks

A well known indirect information flow arising in standard concurrent languages is the so-called *(non)termination leak*. This kind of leak occurs when the result of a fine-grained observation of the low memory [4] depends on the termination or nontermination of some subprogram, which itself depends on the state of the high memory. Typical examples of nontermination leaks are:

[4] This kind of leak only appears with a fine-grained observation, or with a *termination sensitive* notion of noninterference [25], as otherwise nothing can be observed of nonterminating computations.

(5.6) $(\texttt{while}\ x_H \neq 0\ \texttt{do nil})\ ;\ y_L := y_L + 1$

(5.7) $(\texttt{if}\ x_H \neq 0\ \texttt{then}\ (\texttt{loop nil})\ \texttt{else nil})\ ;\ y_L := y_L + 1$

In both these examples the increment of the low variable y_L will take place if and only if the preceding program does not diverge, that is, if and only if the value of the high variable x_H is equal to 0. Note that the possibility of divergence occurs in the first component of the sequential composition, while the low memory change occurs in the second component. Indeed, this kind of leak depends on the occurrence or not of the *passage of control* in the sequential composition statement. Note that there would be no difference if the low memory change took place later in the program, that is, in subsequent sequential components.

A first proposal to rule out nontermination leaks such as those of programs (5.6) and (5.7) was put forward in [24] and adopted by a number of other authors, e.g. [22,1]. The solution consisted in forbidding high loops (that is, loops with a high condition) completely, and forbidding loops in the branches of high conditionals. A second proposal, presented in [23] and independently in [14], was based on the observation that, in the above examples, the high loop and high conditional are not insecure as such (since they do not affect the low memory) but only as far as they are followed, in sequence, by changes in the low memory. The suggested solution was then to forbid the sequential composition $(P\,;\,Q)$ whenever P contains a high loop or high conditional and Q performs low memory changes. This solution has the advantage of being less restrictive than the previous one as concerns the use of high loops, and to be robust with respect to the introduction of scheduling.

Note that programs (5.6) and (5.7) contain instantaneous loops. Indeed, nontermination leaks would not arise in a restricted language where only well-behaved reactive programs could be defined.

Busy waiting leaks

Another kind of leak associated with high loops in a concurrent setting, which is closely related to the previous one, is the *busy waiting leak*. Such leaks are not usually distinguished from nontermination leaks, on the grounds that they originate from similar programs. Here we choose to treat busy waiting leaks separately, because they do not involve divergence. We recall from the Introduction the following example of busy waiting leak for asynchronous parallelism. Consider the program $P \parallel Q$, where $\parallel$ stands for nondeterministic interleaving and P and Q are given by:

Example 7 (*Busy waiting with asynchronous concurrency)*

$$P \,:\, (\texttt{while}\ x_H \neq 0\ \texttt{do nil})\ ;\ y_L := 0\ ;\ x_H := 1$$

$$Q \,:\, (\texttt{while}\ x_H = 0\ \texttt{do nil})\ ;\ y_L := 1\ ;\ x_H := 0$$

Note that this program always terminates, so the issue of termination versus nontermination does not arise here. However, this program is insecure, since it produces different values for y_L depending on the initial value of x_H. The insecurity stems from the same situation exhibited in program (5.6) above, namely the sequential composition between a high loop and a low assignment. Not surprisingly, this kind of leak can be prevented by the same restrictions used to cope with nontermination leaks.

A natural question to ask is whether such busy waiting leaks can also occur in a reactive setting. Indeed, while in an asynchronous model a busy waiting loop can be unblocked by a parallel thread, as shown by the above example, in the reactive model a purely imperative loop cannot be unblocked by another thread, since an executing thread releases the control only by terminating or suspending. Hence, loops cannot give rise to busy waiting leaks in a reactive scenario. In fact, one of the points of reactive programming is precisely to eliminate busy waiting in favour of suspension, which is deemed to be a healthier status in that it allows other threads to proceed or a new instant to start. However, a sort of "light" busy waiting behaviour can still be encoded in our language. Consider the following program, which exhibits a *healthy loop* whose body suspends at each iteration:

$$(5.8)\quad ((\texttt{while}\ x_H \neq 0\ \texttt{do pause})\,;\, y_L := 0)\ \curlywedge\ (y_L := 1\,;\, x_H := 0)$$

Here the first thread suspends, by executing the `pause` statement, 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 this program is insecure. This means that even in a restricted language with no instantaneous loops (and hence no nontermination leaks), high loops may be dangerous if they are sequentially followed by low memory changes.

Suspension leaks

We come now to a kind of insecure flow which is specific to our reactive setting, since it originates from the possibility of suspension. This sort 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, or their persistence after an instant change, possibly leading to insecure flows if these threads perform low memory changes. We already saw some simple examples of suspension leaks at page 25, caused by `when` or `watching` commands with a high control signal.

Another kind of suspension leak, associated with the passage of control in the sequential composition, may be observed in the following program (a variant of example (5) seen in the Introduction):

(5.9) $(\texttt{when}\ a_H\ \texttt{do nil})\,;\ y_L := y_L + 1$

This program is insecure because if a_H is present it increments y_L and if a_H is absent it suspends, performing no observable memory change. Note that we can easily turn this program into a terminating one by surrounding it with a `watching` statement whose control signal is present. Let us also point out the similarity between the suspension leak in (5.9) and the nontermination leak in (5.6), although the two programs behave very differently when plugged in `watching` or $\curlywedge$ contexts. This example shows that a secure `when` statement with a high control signal should not be followed in sequence by a low memory change. A similar example could be given for the `watching` statement.

Note that Example (5.8) above is also an example of suspension leak, where the insecurity originates from the possibility of executing the `pause` statement. A simpler example is the following, where the increment on y_L will be performed if and only a_H is absent:

(5.10) $\texttt{do}\ (\texttt{pause}\,;\ y_L := y_L + 1)\ \texttt{watching}\ a_H$

On the basis of Examples (5.1)–(5.10), we may conclude that, although the phenomena involved are slightly different, the rule of thumb for typing reactive programs will be similar to that for parallel programs in [23,14]. This rule 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. In particular, the rule for typing sequential composition will require that the level of tests in the first component be lower than or equal to the level of writes in the second. We shall call this condition, for short, the *BCS-condition* (from the names of its promotors).

However, there is a further element to consider. Let us look at a more elaborate example, which obeys the BCS-condition ("no low writes after high tests") and yet exhibits an intriguing suspension leak. Here the culprit appears to be the combination of suspension with our particular scheduling.

Example 8 (Scheduling leak)

Consider the program $(P \curlywedge Q) \curlywedge R$*, running in two different signal environments* $E_1 = \{a_H, b_H\}$ *and* $E_2 = \{b_H\}$*:*

$$P : \texttt{pause}\,;\, x_L := 1$$

$$Q : \texttt{do}\ (\texttt{when}\ a_H\ \texttt{do}\ \texttt{nil})\ \texttt{watching}\ b_H \curlywedge \texttt{when}\ c_L\ \texttt{do}\ x_L := 0$$

$$R : \texttt{pause}\,;\, \texttt{emit}\ c_L$$

Here threads P and Q contain different assignments to x_L. Thread P starts executing and suspends after a few steps. Now thread Q 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 P and Q 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 R 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 $\texttt{emit}\ c_L \curlywedge (\texttt{when}\ c_L\ \texttt{do}\ x_L := 0 \curlywedge x_L := 1)$ or in the state $\texttt{emit}\ c_L \curlywedge (x_L := 1 \curlywedge (\texttt{nil} \curlywedge \texttt{when}\ c_L\ \texttt{do}\ x_L := 0))$, depending on whether the starting environment was E_1 or E_2. In any case signal c_L is emitted and we are left either with the process $(\texttt{when}\ c_L\ \texttt{do}\ x_L := 0 \curlywedge x_L := 1)$ or with the process $(x_L := 1 \curlywedge (\texttt{nil} \curlywedge \texttt{when}\ c_L\ \texttt{do}\ x_L := 0))$. In the first case the assignment $x_L := 0$ is executed first, while in the second it is executed after $x_L := 1$.

This example shows that a scheduling leak (which we may consider as a case of suspension leak) may be caused by the coexistence of a high test in a thread with a low write in another thread [5]. This will lead us to impose conditions in the typing rules for the operator $\curlywedge$, which are similar to those required for sequential composition in [14,23], demanding that the level of tests in one component be lower than or equal to the level of writes in the other component. Moreover in the case of $P \curlywedge Q$ this will have to hold in both directions, since the roles of P and Q may be interchanged during execution.

[5] Note that this problem would not arise with a nondeterministic scheduler, which, however, would cause other kinds of leaks which do not occur here, as shown for instance in [23,14].

Types and typing rules

Let us now present our security type system. As mentioned in Section 2, expressions are typed with *simple types*, which are just security levels δ, θ, σ. As usual, these are defined 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. To keep full generality we shall not assume our lattice to be finite nor complete (although the existence of bottom and top elements would allow us to simplify some of the typing rules). Starting from simple types we build *variable types* of the form δ *var* and *signal types* of the form δ *sig*, collectively called *name types*. We shall use δ *name* to denote either δ *var* or δ *sig*.

Program types have the form (θ, σ) *cmd*, as in [14,23]. 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 [14,23]. 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, the introduction of side conditions similar to those for sequential composition is motivated by Example 8 above.

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, like those of [24,22]) in the context of a co-routine mechanism. However, it may be worth examining if this restriction could be lifted by means of techniques proposed for other concurrent languages ([19,28]). We also conjecture that this restriction could be removed by adapting our scheduling policy so that the "scheduler counter" points to the same thread at the beginning of each instant (or rather, to the first created thread among the alive ones).

As an application of our type system, let us reconsider the service-replacement example from Section 2. Assume our lattice is $\{L, H\}$, with $L \leq H$. Suppose the time of replacement t is a sensitive information. Then t will have security level H. Now, as we have seen, the final value of $count_1$ depends on t. This means that $count_1$ should not have security level L, otherwise the system would be insecure. We let the reader verify that, in the hypothesis that t has type $H\ var$, the type system requires $count_1$ to have type $H\ var$ too.

(NIL) $\Gamma \vdash \texttt{nil} : (\theta, \sigma)\ cmd$

(ASSIGN) $$\frac{\Gamma \vdash e : \theta \quad \Gamma(x) = \theta\ var}{\Gamma \vdash x := e : (\theta, \sigma)\ cmd}$$

(LET) $$\frac{\Gamma \vdash e : \delta \qquad \{x : \delta\ var\}\Gamma \vdash P : (\theta, \sigma)\ cmd}{\Gamma \vdash \texttt{let}\ x : \delta = e\ \texttt{in}\ P : (\theta, \sigma)\ cmd}$$

(SEQ) $$\frac{\Gamma \vdash P : (\theta_1, \sigma_1)\ cmd \quad \Gamma \vdash Q : (\theta_2, \sigma_2)\ cmd \quad \sigma_1 \leq \theta_2}{\Gamma \vdash P\,;\, Q : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd}$$

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

(WHILE) $$\frac{\Gamma \vdash e : \delta \quad \Gamma \vdash P : (\theta, \sigma)\ cmd \quad \delta \vee \sigma \leq \theta}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ P : (\theta, \delta \vee \sigma)\ cmd}$$

(EMIT) $$\frac{\Gamma(a) = \theta\ sig}{\Gamma \vdash \texttt{emit}\ a : (\theta, \sigma)\ cmd}$$

(LOCAL) $$\frac{\{a : \delta\ sig\}\Gamma \vdash P : (\theta, \sigma)\ cmd}{\Gamma \vdash \texttt{local}\ a : \delta\ \texttt{in}\ P : (\theta, \sigma)\ cmd}$$

(WATCH) $$\frac{\Gamma(a) = \delta\ sig \quad \Gamma \vdash P : (\theta, \sigma)\ cmd \quad \delta \leq \theta}{\Gamma \vdash \texttt{do}\ P\ \texttt{watching}\ a : (\theta, \delta \vee \sigma)\ cmd}$$

(WHEN) $$\frac{\Gamma(a) = \delta\ sig \quad \Gamma \vdash P : (\theta, \sigma)\ cmd \quad \delta \leq \theta}{\Gamma \vdash \texttt{when}\ a\ \texttt{do}\ P : (\theta, \delta \vee \sigma)\ cmd}$$

(PAR) $$\frac{\Gamma \vdash P : (\theta_1, \sigma_1)\ cmd \quad \Gamma \vdash Q : (\theta_2, \sigma_2)\ cmd \quad \sigma_1 \leq \theta_2 \quad \sigma_2 \leq \theta_1}{\Gamma \vdash P \leftcurvearrow Q : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd}$$

(SUB) $$\frac{\Gamma \vdash P : (\theta, \sigma)\ cmd \quad \theta \geq \theta' \quad \sigma \leq \sigma'}{\Gamma \vdash P : (\theta', \sigma')\ cmd}$$

(EXPR) $$\frac{\forall x_i \in \texttt{fv}(e).\delta \geq \theta_i \text{ where } \Gamma(x_i) = \theta_i\ var}{\Gamma \vdash e : \delta}$$

Fig. 5. Typing Rules

In this section we prove some important properties of our type system, namely *subject reduction*, *guard safety* and *confinement*. The first is the classical type preservation property, stating that types are preserved by execution, while the last two properties formalise the intended meaning of types.

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 $\Gamma \vdash P : (\theta, \sigma)$ *cmd and* $\Gamma' \supseteq \Gamma$, *then* $\Gamma' \vdash P : (\theta, \sigma)$ *cmd.*

(2) If $\{n : \delta \ name\}\Gamma \vdash P : (\theta, \sigma)$ *cmd and* $n' \notin \mathtt{dom}(\Gamma)$, *then* $\{n' : \delta \ name\}\Gamma \vdash \{n'/n\}P : (\theta, \sigma)$ *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. The proofs of the next two results are quite standard, but we include them for completeness.

Lemma 3.2 (Instant changes preserve types)

If $\langle E, P\rangle\ddagger$ *and* $\Gamma \vdash P : (\theta, \sigma)$ *cmd, then* $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)$ *cmd.*

Proof By induction on the proof of $\Gamma \vdash P : (\theta, \sigma)$ *cmd*. We only have to consider the cases of suspendable processes (see Figure 1), corresponding to the typing rules (WHEN), (WATCH), (SEQ), (PAR) and to the subtyping rule (SUB).

- (WHEN) Here $P = \mathtt{when}\ a\ \mathtt{do}\ P_1$ and $\Gamma \vdash P : (\theta, \sigma)$ *cmd* is deduced from $\Gamma(a) = \delta$ *sig*, $\Gamma \vdash P_1 : (\theta, \sigma_1)$ *cmd*, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma_1$. If $a \notin E$ we conclude immediately since $\lfloor P \rfloor_E = P$. If $a \in E$ then $\lfloor P \rfloor_E = \mathtt{when}\ a\ \mathtt{do}\ \lfloor P_1 \rfloor_E$. By induction $\Gamma \vdash \lfloor P_1 \rfloor_E : (\theta, \sigma_1)$ *cmd*, hence, using rule (WHEN) again, we deduce $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)$ *cmd*.
- (WATCH) Here $P = \mathtt{do}\ P_1\ \mathtt{watching}\ a$ and $\Gamma \vdash P : (\theta, \sigma)$ *cmd* is deduced again from $\Gamma(a) = \delta$ *sig*, $\Gamma \vdash P_1 : (\theta, \sigma_1)$ *cmd*, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma_1$. If $a \in E$ we have $\lfloor P \rfloor_E = \mathtt{nil}$ and we can conclude immediately by rule (NIL). If $a \notin E$ then $\lfloor P \rfloor_E = \mathtt{do}\ \lfloor P_1 \rfloor_E\ \mathtt{watching}\ a$. By induction we have $\Gamma \vdash \lfloor P_1 \rfloor_E : (\theta, \sigma_1)$ *cmd*, hence $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)$ *cmd* by rule (WATCH).
- (SEQ) Here $P = P_1 ; P_2$ and $\Gamma \vdash P : (\theta, \sigma)$ *cmd* is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1)$ *cmd*, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)$ *cmd*, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$. We have $\lfloor P_1 ; P_2 \rfloor_E = \lfloor P_1 \rfloor_E ; P_2$. By induction $\Gamma \vdash \lfloor P_1 \rfloor_E : (\theta_1, \sigma_1)$ *cmd*, hence $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)$ *cmd* by rule (SEQ).

- (PAR) Here $P = P_1 \upharpoonleft P_2$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1)\ cmd$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)\ cmd$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$, $\sigma_2 \leq \theta_1$. We have $\lfloor P_1 \upharpoonleft P_2 \rfloor_E = \lfloor P_1 \rfloor_E \upharpoonleft \lfloor P_2 \rfloor_E$. By induction $\Gamma \vdash \lfloor P_i \rfloor_E : (\theta_i, \sigma_i)\ cmd$, hence $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)\ cmd$ by rule (PAR).
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ for some θ', σ' such that $\theta \geq \theta'$ and $\sigma \leq \sigma'$. By induction $\Gamma \vdash \lfloor P \rfloor_E : (\theta', \sigma')\ cmd$, hence, using rule (SUB) again, we deduce $\Gamma \vdash \lfloor P \rfloor_E : (\theta, \sigma)\ cmd$.

□

Theorem 3.3 (Subject reduction)

If $\Gamma \vdash P : (\theta, \sigma)\ cmd$ *and* $\langle \Gamma, S, E, P \rangle \longmapsto \langle \Gamma', S', E', P' \rangle$ *then* $\Gamma' \vdash P' : (\theta, \sigma)\ cmd$.

Proof Let $C = \langle \Gamma, S, E, P \rangle$ and $C' = \langle \Gamma', S', E', P' \rangle$. We want to show that $\Gamma' \vdash P' : (\theta, \sigma)\ cmd$. 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 then conclude by Lemma 3.2.

(2) *Simple move.* Suppose now $C \rightarrow C'$. We show that $\Gamma' \vdash P' : (\theta, \sigma)\ cmd$ by induction on the proof of $\Gamma \vdash P : (\theta, \sigma)\ cmd$. We examine the cases where P is not terminated nor suspended. Note that in the cases of rules (ASSIGN) and (EMIT) we have $P' = \texttt{nil}$ and thus we can conclude immediately using rule (NIL). We consider some of the other cases.
   - (LET) Here $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ P_1$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash e : \delta$ and $\{x : \delta\ var\}\Gamma \vdash P_1 : (\theta, \sigma)\ cmd$. Since $C \rightarrow C'$ is derived by (LET-OP), we have $\Gamma' = \{x' : \delta\ var\}\Gamma$ and $P' = \{x'/x\}P_1$ for some $x' \notin \texttt{dom}(\Gamma)$. Then by Proposition 3.1 we can conclude that $\{x' : \delta\ var\}\Gamma \vdash \{x'/x\}P_1 : (\theta, \sigma)\ cmd$.
   - (SEQ) Here $P = P_1 ; P_2$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash P_1 : (\theta_1, \sigma_1)\ cmd$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)\ cmd$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$.
     - If $P_1 = \texttt{nil}$, then $C \rightarrow C'$ is derived by (SEQ-OP$_1$) 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 $\Gamma' \vdash P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd$.
     - If $P_1 \neq \texttt{nil}$ then $C \rightarrow C'$ is derived by (SEQ-OP$_2$) 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 $\Gamma' \vdash P_1' : (\theta_1, \sigma_1)\ cmd$. By Proposition 2.1 $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Gamma')$, hence by Proposition 3.1 we get $\Gamma' \vdash P_2 : (\theta_2, \sigma_2)\ cmd$. Then by rule (SEQ) we obtain $\Gamma' \vdash P_1' ; P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd$.
   - (COND) Here $P = \texttt{if}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash e : \delta$, $\Gamma \vdash P_1 : (\theta, \sigma')\ cmd$ and $\Gamma \vdash P_2 : (\theta, \sigma')\ cmd$ where $\sigma = \delta \vee \sigma'$.

- If $S(e) = \texttt{true}$, then $C \to C'$ is derived using rule (COND-OP$_1$) and we have $\Gamma' = \Gamma$ and $P' = P_1$. Since $\Gamma' \vdash P_1 : (\theta, \sigma')\ cmd$ and $\sigma \geq \sigma'$, by (SUB) we have $\Gamma' \vdash P_1 : (\theta, \sigma)\ cmd$.
- The case $S(e) = \texttt{false}$ is symmetric.

• (WHILE) Here $P = \texttt{while}\ e\ \texttt{do}\ P_1$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash e : \delta$, $\Gamma \vdash P_1 : (\theta, \sigma')\ cmd$ and $\delta \vee \sigma' \leq \theta$ where $\sigma = \delta \vee \sigma'$.
- If $S(e) = \texttt{true}$, then $C \to C'$ is derived using rule (WHILE-OP$_1$) and we have $\Gamma' = \Gamma$ and $P' = P_1\,;\ \texttt{while}\ e\ \texttt{do}\ P_1$. Since $\Gamma \vdash P_1 : (\theta, \sigma')\ cmd$ and $\sigma \geq \sigma'$, by (SUB) we have $\Gamma \vdash P_1 : (\theta, \sigma)\ cmd$. Since $\sigma \leq \theta$, we may then use (SEQ) to deduce $\Gamma' \vdash P' : (\theta, \sigma)\ cmd$.
- If $S(e) = \texttt{false}$, then $C \to C'$ is derived using rule (WHILE-OP$_2$). Then $\Gamma' = \Gamma$ and $P' = \texttt{nil}$, and we conclude by rule (NIL).

• (WATCH) Here $P = \texttt{do}\ P_1\ \texttt{watching}\ a$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma(a) = \delta\ sig$, $\Gamma \vdash P_1 : (\theta, \sigma')\ cmd$ and $\delta \leq \theta$, where $\sigma = \delta \vee \sigma'$.
- If $P_1 = \texttt{nil}$, then $C \to C'$ is derived using rule (WATCH-OP$_1$). Then $\Gamma' = \Gamma$ and $P' = \texttt{nil}$, and we conclude by rule (NIL).
- If $P_1 \neq \texttt{nil}$, then $C \to C'$ is derived using rule (WATCH-OP$_2$) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \to \langle \Gamma', S', E', P_1' \rangle$. In this case $P' = \texttt{do}\ P_1'\ \texttt{watching}\ a$. By induction $\Gamma' \vdash P_1' : (\theta, \sigma')\ cmd$, so by (WATCH) again we deduce $\Gamma' \vdash \texttt{do}\ P_1'\ \texttt{watching}\ a : (\theta, \sigma)\ cmd$.

• (PAR) Here $P = P_1 \curlywedge P_2$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P_1 : (\theta_1, \sigma_1)\ cmd$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)\ cmd$, $\theta = \theta_1 \wedge \theta_2$, $\sigma = \sigma_1 \vee \sigma_2$, $\sigma_1 \leq \theta_2$, $\sigma_2 \leq \theta_1$.
- If $P_1 = \texttt{nil}$, then $C \to C'$ is derived using rule (PAR-OP$_1$). 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) $\Gamma' \vdash P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd$.
- If $P_1 \neq \texttt{nil}$ and $\neg \langle E, P \rangle_1 \ddagger$ then $C \to C'$ is derived using rule (PAR-OP$_2$) from the hypothesis $\langle \Gamma, S, E, P_1 \rangle \to \langle \Gamma', S', E', P_1' \rangle$. We then have $P' = P_1' \curlywedge P_2$. By induction $\Gamma' \vdash P_1' : (\theta_1, \sigma_1)\ cmd$. By Proposition 2.1 $\texttt{dom}(\Gamma) \subseteq \texttt{dom}(\Gamma')$, hence we may use Proposition 3.1 to get $\Gamma' \vdash P_2 : (\theta_2, \sigma_2)\ cmd$. Then, using rule (PAR) again, we obtain $\Gamma' \vdash P_1' \curlywedge P_2 : (\theta_1 \wedge \theta_2, \sigma_1 \vee \sigma_2)\ cmd$.
- If $P_1 \neq \texttt{nil}$ and $\langle E, P \rangle_1 \ddagger$ then $C \to C'$ is derived by rule (PAR-OP$_3$) and we have $C' = \langle \Gamma, S, E, P_2 \curlywedge P_1 \rangle$. 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 $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ for some θ', σ' such that $\theta \geq \theta'$ and $\sigma \leq \sigma'$. By induction $\Gamma' \vdash P' : (\theta', \sigma')\ cmd$, hence by rule (SUB) we conclude that $\Gamma' \vdash P' : (\theta, \sigma)\ cmd$.

□

Our next result ensures that program types have the intended meaning. Let us first introduce some terminology. A program P is said to be in *standard form* (or standard) if all bound names of P are distinct and no name is simultaneously free and bound in P. Note that every program can be put into standard form using α-conversion. For P in standard form, we use the generic term *guard* to denote any variable appearing in the condition of a loop or of a conditional in P, and any signal controlling a `when` or `watching` statement in P. Formally, if $\mathtt{var}(e)$ is the set of variables occurring in the expression e, the guards of P are defined as follows:

Definition 3.1 (Guard)

The set of guards *of a standard program P, denoted* $\mathtt{guards}(P)$*, is defined inductively as follows:*

- $\mathtt{guards}(\mathtt{nil}) = \mathtt{guards}(x := e) = \mathtt{guards}(\mathtt{emit}\ a) = \emptyset$;
- $\mathtt{guards}(\mathtt{let}\ x : \delta = e\ \mathtt{in}\ Q) = \mathtt{guards}(\mathtt{local}\ a : \delta\ \mathtt{in}\ Q) = \mathtt{guards}(Q)$;
- $\mathtt{guards}(\mathtt{if}\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) = \mathtt{var}(e) \cup \mathtt{guards}(P_1) \cup \mathtt{guards}(P_2)$;
- $\mathtt{guards}(\mathtt{while}\ e\ \mathtt{do}\ Q) = \mathtt{var}(e) \cup \mathtt{guards}(Q)$;
- $\mathtt{guards}(\mathtt{do}\ Q\ \mathtt{watching}\ a) = \mathtt{guards}(\mathtt{when}\ a\ \mathtt{do}\ Q) = \{a\} \cup \mathtt{guards}(Q)$;
- $\mathtt{guards}(Q \leftarrow R) = \mathtt{guards}(Q\,;\,R) = \mathtt{guards}(Q) \cup \mathtt{guards}(R)$;

For instance, x is a guard in the program $(\mathtt{while}\ x \leq y\ \mathtt{do}\ y := y - 1)$ and a is a guard in the program $x := 0\,;(\mathtt{when}\ a\ \mathtt{do}\ x := 1)$.

Moreover, if P is in standard form and P is typable in the type environment Γ, 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 $(x, \delta\ var)$ (resp. $(a, \delta\ sig)$).

ii) x (resp. a) is bound in P by a local declaration $\mathtt{let}\ x : \delta = e\ \mathtt{in}\ Q$ (resp. $\mathtt{local}\ a : \delta\ \mathtt{in}\ Q$).

Note that for case *ii)* the environment Γ is actually immaterial.

Lemma 3.4 (Guard safety and confinement)

(1) If $\Gamma \vdash P : (\theta, \sigma)$ cmd then every guard in P has a security level δ in Γ such that $\delta \leq \sigma$.

(2) If $\Gamma \vdash P : (\theta, \sigma)$ cmd then every written variable or emitted signal in P has a security level δ in Γ 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): these cases correspond to imperative constructs. The proof is the same as in [14] and is thus omitted. We examine some of the remaining cases.
- (LET) Here $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$, with $\{x : \delta\ var\}\Gamma \vdash Q : (\theta, \sigma)\ cmd$ and $\Gamma \vdash e : \delta$. By induction every guard in Q has a security level $\delta' \leq \sigma$ in the type environment $\{x : \delta\ var\}\Gamma$. Then every guard of P different from x has a 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 $\delta \leq \sigma$.
- (WATCH) Here $P = \texttt{do}\ Q\ \texttt{watching}\ a$, with $\Gamma(a) = \delta\ sig$, $\Gamma \vdash Q : (\theta, \sigma')\ cmd$ and $\sigma = \delta \vee \sigma'$. By induction every guard in Q has a 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 $\delta \leq \sigma$.
- (PAR) Here $P = P_1 \triangleleft P_2$ 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 a level $\delta_i \leq \sigma_i$. Since $\sigma_i \leq \sigma_1 \vee \sigma_2$ we can then conclude.

Proof of (2). By induction on the inference of $\Gamma \vdash P : (\theta, \sigma)\ cmd$. The proof for (NIL), (ASSIGN), (SEQ), (COND), (WHILE), (SUB) is as in [14]. We consider some of the other cases.

- (LET) Here $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$, with $\{x : \delta\ var\}\Gamma \vdash Q : (\theta, \sigma)\ cmd$ and $\Gamma \vdash e : \delta$. By induction every written variable or emitted signal in Q has a security level δ' such that $\theta \leq \delta'$ in the type environment $\{x : \delta\ var\}\Gamma$. Then every written variable or emitted signal different from x in P has a 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 $\theta \leq \delta$.
- (EMIT) Here $P = \texttt{emit}\ a$, and $\Gamma(a) = \theta\ sig$. This case is trivial since the only emitted signal has security level θ.
- (WATCH) Here $P = \texttt{do}\ Q\ \texttt{watching}\ a$, with $\Gamma(a) = \delta\ sig$ and $\Gamma \vdash Q : (\theta, \sigma')\ cmd$. By induction every written variable or emitted signal in Q has a security level δ' such that $\theta \leq \delta'$. Whence the conclusion, since P does not introduce any written variables nor emitted signals.
- (PAR) Here $P = P_1 \triangleleft P_2$ with $\Gamma \vdash P_1 : (\theta_1, \sigma_1)\ cmd$, $\Gamma \vdash P_2 : (\theta_2, \sigma_2)\ cmd$ and $\theta = \theta_1 \wedge \theta_2$. By induction every written variable or emitted signal in P_i has a security level δ_i with $\theta_i \leq \delta_i$. Since $\theta_1 \wedge \theta_2 \leq \theta_i$ we can then conclude.

□

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, that is, security levels in $\mathcal{L}$.

Two memories are said to be low-equal if their low parts coincide. This is formally defined as follows:

Definition 3.2 ($\mathcal{L}, \Gamma$-equality of stores and signal environments)
Let S_1, S_2 be variable stores, E_1, E_2 be signal environments, $\mathcal{L}$ be a downward-closed set of security levels and Γ be a type environment.

The low equality $=^{\Gamma}_{\mathcal{L}}$ *on stores and signal environments is defined by:*

$S_1 =^{\Gamma}_{\mathcal{L}} S_2$ *if for any* $x \in \mathtt{dom}(\Gamma)$:

$$(\Gamma(x) = \theta\ \mathit{var}\ \wedge\ \theta \in \mathcal{L})\ \Rightarrow\ (\,(x \in \mathtt{dom}(S_1)\ \Leftrightarrow x\ \in \mathtt{dom}(S_2))\ \wedge\ (x \in \mathtt{dom}(S_i)\ \Rightarrow\ S_1(x) = S_2(x))\,)$$

$E_1 =^{\Gamma}_{\mathcal{L}} E_2$ *if for any* $a \in \mathtt{dom}(\Gamma)$:

$$(\Gamma(a) = \theta\ \mathit{sig}\ \wedge\ \theta \in \mathcal{L})\ \Rightarrow\ (\,a \in E_1 \Leftrightarrow a \in E_2)$$

Definition 3.3 ($\mathcal{L}, \Gamma$-equality of memories)
Let S_1, S_2 be variable stores, E_1, E_2 be signal environments, $\mathcal{L}$ be a downward-closed set of security levels and Γ be a type environment.

The low equality $=^{\Gamma}_{\mathcal{L}}$ *on memories is defined by:*

$$\langle S_1, E_1\rangle =^{\Gamma}_{\mathcal{L}} \langle S_2, E_2\rangle \quad \mathit{if} \quad S_1 =^{\Gamma}_{\mathcal{L}} S_2 \ \ \mathit{and} \ \ E_1 =^{\Gamma}_{\mathcal{L}} E_2$$

There is a class of programs for which the security property is particularly easy to establish because of their inability to change the low memory. 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 3.4 (High programs)

Let Γ be a type environment and $\mathcal{L}$ be a downward-closed set of security levels.

1. Syntactically high programs

The set $\mathcal{H}^{\Gamma,\mathcal{L}}_{\text{syn}}$ of syntactically high programs is inductively defined by:
$P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\text{syn}}$ if one of the following holds:

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

2. Semantically high programs

The set $\mathcal{H}^{\Gamma,\mathcal{L}}_{\text{sem}}$ of semantically high programs is coinductively defined by:
$P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\text{sem}}$ implies that for any variable store S and signal environment E, both of the following hold:

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

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 $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$ (as well as $P = \texttt{local}\ a : \delta\ \texttt{in}\ Q$) 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 $(\texttt{local}\ a : \delta\ \texttt{in}\ \texttt{nil})$ is semantically high while $(\texttt{local}\ a : \delta\ \texttt{in}\ \texttt{emit}\ a)$ is not. The second clause of Definition 3.4.2 concerns instant changes. As argued in the Introduction, we do not consider as observable the reset of the low sig-

nal environment that is induced by instant changes. This is reflected by the absence of the low equality condition in the second clause of Definition 3.4.2 (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 1 *For any* Γ *and for any downward-closed set* $\mathcal{L} \subseteq \mathcal{T}$, $\mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{syn}} \subseteq \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{sem}}$.

As can be expected, the converse is not true. An example of a semantically high program that is not syntactically high is (`if true then nil else` $y_L := 0$).

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

Fact 2 *Let* Γ *be a type environment and* $\mathcal{L}$ *be a downward-closed set of security levels. If for some* $\theta \notin \mathcal{L}$ *there exists* σ *such that* $\Gamma \vdash P : (\theta, \sigma)\ cmd$, *then* $P \in \mathcal{H}^{\Gamma,\mathcal{L}}_{\mathtt{syn}}$.

Proof Immediate, by the Confinement lemma (Lemma 3.4.2). □

We introduce now the notion of $\mathcal{L}$-Guardedness, borrowed from [14]. This formalises the property of programs containing no high guards.

Definition 3.5 ($\mathcal{L}$-guardedness)
Let Γ *be a type environment and* $\mathcal{L}$ *be a downward-closed set of security levels. A program* P *is* $\mathcal{L}$-guarded *in* Γ *if for some* θ, *there exists* $\sigma \in \mathcal{L}$ *such that* $\Gamma \vdash P : (\theta, \sigma)\ cmd$.

Fact 3 *If* P *is* $\mathcal{L}$-guarded *in* Γ *then every guard in* P *has a security level* δ *in* Γ *such that* $\delta \in \mathcal{L}$.

Proof Immediate, by the Guard safety lemma (Lemma 3.4.1). □

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 $\Gamma \vdash P : (\theta, \sigma)\ cmd$.

We now proceed to prove two results which will be the basis of our soundness proof. The proofs will therefore be presented in full detail. The first result concerns $\mathcal{L}$-guarded programs: it states that such programs, when run in low-equal memories, produce at each step equal type environments and programs, and low-equal memories.

Theorem 3.5 (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* $\langle \Gamma, S_2, E_2, P\rangle \hookrightarrow \langle \Gamma', S_2', E_2', P'\rangle$ *and* $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$.

(2) (Simple move) $\langle \Gamma, S_1, E_1, P\rangle \rightarrow \langle \Gamma', S_1', E_1', P'\rangle$ *implies* $\langle \Gamma, S_2, E_2, P\rangle \rightarrow \langle \Gamma', S_2', E_2', P'\rangle$ *and* $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$.

Proof

Proof of (1). By induction on the inference of $\Gamma \vdash P : (\theta, \sigma)\ cmd$ for $\sigma \in \mathcal{L}$, 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 1), 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 $\langle \Gamma, S_1, E_1, P\rangle \hookrightarrow \langle \Gamma, S_1, \emptyset, \lfloor P\rfloor_{E_1}\rangle$ and $\langle \Gamma, S_2, E_2, P\rangle \hookrightarrow \langle \Gamma, S_2, \emptyset, \lfloor P\rfloor_{E_2}\rangle$, 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 $P = \texttt{when}\ a\ \texttt{do}\ Q$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma(a) = \delta\ sig$, $\delta \leq \theta$ 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:
  - $\langle E_1, P\rangle\ddagger$ is deduced by rule (WHEN-SUS$_1$): in this case $a \notin E_1$. Note that $\delta \leq \sigma$ implies $\delta \in \mathcal{L}$ and thus, since $E_1 =_{\mathcal{L}}^{\Gamma} E_2$, $a \in E_1 \Leftrightarrow a \in E_2$. Then also $a \notin E_2$ and by rule (WHEN-SUS$_1$) we deduce $\langle E_2, P\rangle\ddagger$.
  - $\langle E_1, P\rangle\ddagger$ is deduced by rule (WHEN-SUS$_2$) 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-SUS$_2$) we obtain $\langle E_2, P\rangle\ddagger$.
- (WATCH), (SEQ), (PAR) By straightforward induction as in the second case of (WHEN).
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ 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 $\Gamma \vdash P : (\theta, \sigma)\ cmd$ where $\sigma \in \mathcal{L}$. We examine all representative cases.

- (ASSIGN) Here $P = x := e$ with $\Gamma \vdash e : \theta$ and $\Gamma(x) = \theta\, var$. By rule (ASSIGN-OP) we then have both $\langle \Gamma, S_1, E_1, P\rangle \rightarrow \langle \Gamma, S_1', E_1, \texttt{nil}\rangle$ and $\langle \Gamma, S_2, E_2, P\rangle \rightarrow \langle \Gamma, S_2', E_2, \texttt{nil}\rangle$, 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 $P = \texttt{let}\ x : \delta = e\ \texttt{in}\ Q$, and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash e : \delta$ and $\{x : \delta\ var\}\Gamma \vdash Q : (\theta, \sigma)\ cmd$, where $\sigma \in \mathcal{L}$. Then $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma', S_1', E_1, P'\rangle$ is inferred by rule (LET-OP), and for some $x' \notin \texttt{dom}(\Gamma)$, we have $\Gamma' = \{x' : \delta\ var\}\Gamma$, $S_1' = \{x' \mapsto S_1(e)\}S_1$ and $P' = \{x'/x\}P$. Then, using rule (LET-OP) again and choosing the same x' we obtain $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma', S_2', E_2, P'\rangle$, 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 fact that $S_1(e) = S_2(e)$ if $\theta \in \mathcal{L}$.
- (SEQ) Here $P = Q\,;\, R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$.
  - If $Q = \texttt{nil}$, then by (SEQ-OP$_1$) we have $\langle \Gamma, S_1, E_1, Q; R\rangle \to \langle \Gamma, S_1, E_1, R\rangle$ and $\langle \Gamma, S_2, E_2, Q; R\rangle \to \langle \Gamma, S_2, E_2, R\rangle$, so we can conclude immediately.
  - If $Q \neq \texttt{nil}$, then $\langle \Gamma, S_1, E_1, Q; R\rangle \to \langle \Gamma', S_1', E_1', P'\rangle$ is derived by rule (SEQ-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$, and we have $P' = Q'; R$. Since $\sigma' \leq \sigma$, Q is $\mathcal{L}$-guarded. We can then use induction to obtain $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. We then have $\langle \Gamma, S_2, E_2, Q; R\rangle \to \langle \Gamma', S_2', E_2', Q'; R\rangle$ by (SEQ-OP$_2$) and we can conclude.
- (COND) Here $P = \texttt{if}\ e\ \texttt{then}\ Q\ \texttt{else}\ R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash e : \delta$, $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma \vdash R : (\theta, \sigma')\ cmd$, 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 $S_i(e) = \texttt{true}$, then by rule (COND-OP$_1$) $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, Q\rangle$ and $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma, S_2, E_2, Q\rangle$ and we can conclude immediately. The case where $S_i(e) = \texttt{false}$ is symmetric.
- (EMIT) Here $P = \texttt{emit}\ a$, where $\Gamma(a) = \theta\ sig$. By rule (EMIT-OP) we have for $i = 1, 2$ the transition $\langle \Gamma, S_i, E_i, \texttt{emit}\ a\rangle \to \langle \Gamma, S_i, E_i', \texttt{nil}\rangle$, 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.
- (WATCH) Here $P = \texttt{do}\ Q\ \texttt{watching}\ a$, with $\Gamma(a) = \delta\ sig$, $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$.
  - $Q = \texttt{nil}$. In this case $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, \texttt{nil}\rangle$ is deduced using rule (WATCH-OP$_1$), and by the same rule $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma, S_2, E_2, \texttt{nil}\rangle$, so we can conclude immediately.
  - $Q \neq \texttt{nil}$. In this case the transition of the first process is of the form $\langle \Gamma, S_1, E_1, \texttt{do}\ Q\ \texttt{watching}\ a\rangle \to \langle \Gamma', S_1', E_1', \texttt{do}\ Q'\ \texttt{watching}\ a\rangle$ and it is derived by rule (WATCH-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$. Note that $\sigma' \leq \sigma$ implies $\sigma' \in \mathcal{L}$, thus Q is also $\mathcal{L}$-guarded. Then by induction $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$, with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Whence

we obtain $\langle \Gamma, S_2, E_2, \texttt{do } Q \texttt{ watching } a\rangle \to \langle \Gamma', S_2', E_2', \texttt{do } Q' \texttt{ watching } a\rangle$ using rule (WATCH-OP$_2$).

- (WHEN) Here $P = \texttt{when } a \texttt{ do } Q$, with $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma(a) = \delta\ sig$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. As in the previous case, there are two possibilities:
  - $Q = \texttt{nil}$. Then $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, \texttt{nil}\rangle$ by rule (WHEN-OP$_1$). Note that $\delta \leq \sigma$ implies $\delta \in \mathcal{L}$ and therefore, since $E_1 =_{\mathcal{L}}^{\Gamma} E_2$, we know that $a \in E_1 \Leftrightarrow a \in E_2$. We know that $a \in E_1$, thus also $a \in E_2$. We can then apply rule (WHEN-OP$_1$) again to get $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma, S_2, E_2, \texttt{nil}\rangle$.
  - $Q \neq \texttt{nil}$. Then $\langle \Gamma, S_1, E_1, \texttt{when } Q \texttt{ do } a\rangle \to \langle \Gamma', S_1', E_1', \texttt{when } Q' \texttt{ do } a\rangle$ is derived by rule (WHEN-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$. Since $\sigma' \leq \sigma$ implies $\sigma' \in \mathcal{L}$, we know that Q is $\mathcal{L}$-guarded. Then by induction $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$, with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Since $a \in E_2$, we have then $\langle \Gamma, S_2, E_2, \texttt{when } Q \texttt{ do } a\rangle \to \langle \Gamma', S_2', E_2', \texttt{when } Q' \texttt{ do } a\rangle$ by rule (WHEN-OP$_2$) and we can conclude.
- (PAR) Here $P = Q \upharpoonleft R$ with $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$ 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:
  - $Q = \texttt{nil}$. Then by rule (PAR-OP$_1$) we have $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, R\rangle$ and $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma, S_2, E_2, R\rangle$, and we can conclude.
  - $Q \neq \texttt{nil}$ and $\neg\langle E_1, Q\rangle\ddagger$. Then $\langle \Gamma, S_1, E_1, Q \upharpoonleft R\rangle \to \langle \Gamma_1, S_1', E_1', Q' \upharpoonleft R\rangle$ is derived using rule (PAR-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma_1, S_1', E_1', Q'\rangle$. By induction $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma_2, S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Then by (PAR-OP$_2$) we deduce $\langle \Gamma, S_2, E_2, Q \upharpoonleft R\rangle \to \langle \Gamma_2, S_2', E_2', Q' \upharpoonleft R\rangle$.
  - $\langle E_1, Q\rangle\ddagger$ and $\neg\langle E_1, R\rangle\ddagger$. Then $\langle \Gamma, S_1, E_1, Q \upharpoonleft R\rangle \to \langle \Gamma, S_1, E_1, R \upharpoonleft Q\rangle$ by (PAR-OP$_3$). 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 rule (PAR-OP$_3$) again to deduce $\langle \Gamma, S_2, E_2, Q \upharpoonleft R\rangle \to \langle \Gamma, S_2, E_2, R \upharpoonleft Q\rangle$ and we conclude.
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ 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 second key result for proving soundness 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 3.5.

Theorem 3.6 (Behaviour of non $\mathcal{L}$-guarded programs)

Let P be typable and non $\mathcal{L}$-Guarded in Γ and suppose $\langle S_1, E_1\rangle =_{\mathcal{L}}^{\Gamma} \langle S_2, E_2\rangle$. 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* $\langle \Gamma, S_2, E_2, P\rangle \hookrightarrow \langle \Gamma', S_2', E_2', P'\rangle$ *and* $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$.

(2) (Simple move) $\langle \Gamma, S_1, E_1, P\rangle \rightarrow \langle \Gamma', S_1', E_1', P'\rangle$ *implies* $\langle \Gamma, S_2, E_2, P\rangle \rightarrow \langle \Gamma', S_2', E_2', P'\rangle$ *and* $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$.

Proof The proof amounts to showing that a non low-guarded program P satisfies the same properties as low-guarded programs, as stated in Theorem 3.5, until it (possibly) becomes syntactically high. We prove the two clauses separately, by induction on the inference of $\Gamma \vdash P : (\theta, \sigma)\ cmd$.

Proof of (1). We start by showing that if $\langle E_1, P\rangle\ddagger$ then either $\langle E_2, P\rangle\ddagger$ (in which case the transition $\langle \Gamma, S_1, E_1, P\rangle \hookrightarrow \langle \Gamma, S_1, \emptyset, \lfloor P\rfloor_{E_1}\rangle$ is matched by $\langle \Gamma, S_2, E_2, P\rangle \hookrightarrow \langle \Gamma, S_2, \emptyset, \lfloor P\rfloor_{E_2}\rangle$ using rule (INSTANT-OP), as in the proof of Theorem 3.5), 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 $P = \texttt{when}\ a\ \texttt{do}\ Q$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma(a) = \delta\ sig$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. There are two cases:
  - $\langle E_1, P\rangle\ddagger$ is deduced by rule (WHEN-SUS$_1$): in this case $a \notin E_1$. Then either $a \notin E_2$ and we can conclude using rule (WHEN-SUS$_1$), 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.
  - $\langle E_1, P\rangle\ddagger$ is deduced by rule (WHEN-SUS$_2$) from the hypothesis $\langle E_1, Q\rangle\ddagger$. By induction either $\langle E_2, Q\rangle\ddagger$, in which case also $\langle E_2, P\rangle\ddagger$ by rule (WHEN-SUS$_2$), or Q is syntactically high. In the latter case by Definition 3.4 also P is syntactically high.
- (WATCH) Easy induction as in the second case of (WHEN).
- (SEQ) Here $P = Q\,;\, R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the following hypotheses: $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$ and $\sigma' \leq \theta''$. In this case $\langle E_1, P\rangle\ddagger$ is deduced by rule (SEQ-SUS) from $\langle E_1, Q\rangle\ddagger$. By induction either $\langle E_2, Q\rangle\ddagger$, in which case $\langle E_2, P\rangle\ddagger$ by rule (SEQ-SUS) again, or Q is syntactically high and, by virtue of Theorem 3.5, 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 3.4, also $Q; R$ is syntactically high.

- (PAR) Here $P = Q \upharpoonleft R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$ and $\sigma'' \leq \theta'$. There are three possibilities:
  - 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 3.4 also $P = Q \upharpoonleft R$ is syntactically high.
  - 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 $\langle E_1, P\rangle\ddagger$ is deduced by rule (PAR-SUS) from $\langle E_1, Q\rangle\ddagger$ and $\langle E_1, R\rangle\ddagger$. Since Q is $\mathcal{L}$-guarded, by Theorem 3.5 we have $\langle E_2, Q\rangle\ddagger$. Since R is not $\mathcal{L}$-guarded we know by induction that either $\langle E_2, R\rangle\ddagger$ or R is syntactically high. If $\langle E_2, R\rangle\ddagger$ we may apply rule (PAR-SUS) to get $\langle E_2, P\rangle\ddagger$. 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 $P = Q \upharpoonleft R$ is syntactically high.
  - 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 $\langle E_1, Q\rangle\ddagger$ and $\langle E_1, R\rangle\ddagger$. Then by Theorem 3.5 we obtain $\langle E_2, Q\rangle\ddagger$ and $\langle E_2, R\rangle\ddagger$. Whence by rule (PAR-SUS) we conclude that $\langle E_2, P\rangle\ddagger$.
- (SUB) Here $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. Then we may conclude immediately, using Theorem 3.5 if $\sigma' \in \mathcal{L}$, and induction otherwise.

Proof of (2). We consider now the case where $\neg\langle E_1, P\rangle\ddagger$. For the four rules (ASSIGN), (LET), (EMIT), (LOCAL) the result is proved exactly in the same way as for Theorem 3.5, 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 $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$.
  - If $Q = \texttt{nil}$, then by (SEQ-OP$_1$) we have $\langle \Gamma, S_1, E_1, Q; R\rangle \to \langle \Gamma, S_1, E_1, R\rangle$ and $\langle \Gamma, S_2, E_2, Q; R\rangle \to \langle \Gamma, S_2, E_2, R\rangle$, so we can conclude immediately.
  - If $Q \neq \texttt{nil}$, then $\langle \Gamma, S_1, E_1, Q; R\rangle \to \langle \Gamma', S_1', E_1', P'\rangle$ is derived by rule (SEQ-OP$_2$) from the hypothesis $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$, and we have $P' = Q'; R$. There are two possibilities:
    - Q is $\mathcal{L}$-guarded. Then by Theorem 3.5 $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Hence by (SEQ-OP$_2$) we can conclude.
    - Q is not $\mathcal{L}$-guarded. By induction we know that either we have $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$, and we conclude using (SEQ-OP$_2$) 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 we know that R is syntactically high, whence by Definition 3.4 also $Q; R$ is syntactically high.

- (COND) Here $P = \texttt{if}\ e\ \texttt{then}\ Q\ \texttt{else}\ R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash e : \delta$, $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma \vdash R : (\theta, \sigma')\ cmd$, 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 3.5. 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 3.4 also P is syntactically high.
- (WATCH) Here $P = \texttt{do}\ Q\ \texttt{watching}\ a$, with $\Gamma(a) = \delta\ sig$, $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. We distinguish two cases:
  - $Q = \texttt{nil}$. In this case $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, \texttt{nil}\rangle$ is deduced by rule (WATCH-OP$_1$) and by the same rule $\langle \Gamma, S_2, E_2, P\rangle \to \langle \Gamma, S_2, E_2, \texttt{nil}\rangle$, so we can conclude immediately.
  - $Q \neq \texttt{nil}$. In this case the transition of the first process is of the form $\langle \Gamma, S_1, E_1, \texttt{do}\ Q\ \texttt{watching}\ a\rangle \to \langle \Gamma', S_1', E_1', \texttt{do}\ Q'\ \texttt{watching}\ a\rangle$ and it is derived by rule (WATCH-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$. There are two subcases:
    i) Q is $\mathcal{L}$-guarded. By Theorem 3.5 $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$, with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Then by rule (WATCH-OP$_2$) we obtain $\langle \Gamma, S_2, E_2, \texttt{do}\ Q\ \texttt{watching}\ a\rangle \to \langle \Gamma', S_2', E_2', \texttt{do}\ Q'\ \texttt{watching}\ a\rangle$.
    ii) Q is not $\mathcal{L}$-guarded. In this case, by induction we know that either $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$ 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 3.4 also P is syntactically high.
- (WHEN) Here $P = \texttt{when}\ a\ \texttt{do}\ Q$, with $\Gamma \vdash Q : (\theta, \sigma')\ cmd$, $\Gamma(a) = \delta\ sig$, $\delta \leq \theta$ and $\sigma = \delta \vee \sigma'$. Assume $a \in E_1$ (otherwise this case is vacuous). There are two possibilities:
  - $Q = \texttt{nil}$. Then $\langle \Gamma, S_1, E_1, P\rangle \to \langle \Gamma, S_1, E_1, \texttt{nil}\rangle$ by rule (WHEN-OP$_1$). If $\delta \in \mathcal{L}$ then $a \in E_1 \Leftrightarrow a \in E_2$ and we proceed as in the proof of Theorem 3.5. 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 3.4 also P is syntactically high.
  - $Q \neq \texttt{nil}$. Then $\langle \Gamma, S_1, E_1, \texttt{when}\ a\ \texttt{do}\ Q\rangle \to \langle \Gamma', S_1', E_1', \texttt{when}\ a\ \texttt{do}\ Q'\rangle$ is derived by rule (WHEN-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \to \langle \Gamma', S_1', E_1', Q'\rangle$. There are two subcases:
    i) Q is $\mathcal{L}$-guarded. Then by Theorem 3.5 $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$, with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. Since $\delta \notin \mathcal{L}$, $a \in E_2$ and by (WHEN-OP$_2$) we may deduce $\langle \Gamma, S_2, E_2, \texttt{when}\ Q\ \texttt{do}\ a\rangle \to \langle \Gamma', S_2', E_2', \texttt{when}\ Q'\ \texttt{do}\ a\rangle$.
    ii) Q is not $\mathcal{L}$-guarded. By induction either Q is syntactically high, in which case by Definition 3.4 we deduce that also P is syntactically high, or $\langle \Gamma, S_2, E_2, Q\rangle \to \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$. In this case, there are two possibilities: if $\delta \in \mathcal{L}$ then $a \in E_2$ and we can apply rule (WHEN-OP$_2$) as in the case where Q is $\mathcal{L}$-guarded; if $\delta \notin \mathcal{L}$, since $\delta \leq \theta$ we have that also $\theta \notin \mathcal{L}$. Then by Lemma 3.4 Q is syntactically high, hence also P is syntactically high.

- (PAR) Here $P = Q \upharpoonleft R$ and $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from the hypotheses $\Gamma \vdash Q : (\theta', \sigma')\ cmd$, $\Gamma \vdash R : (\theta'', \sigma'')\ cmd$, $\theta = \theta' \wedge \theta''$, $\sigma = \sigma' \vee \sigma''$, $\sigma' \leq \theta''$ and $\sigma'' \leq \theta'$. We distinguish two cases, depending on whether $\langle E_1, Q\rangle\ddagger$ or $\neg\langle E_1, Q\rangle\ddagger$.

  Suppose first $\neg\langle E_1, Q\rangle\ddagger$. There are two possibilities:

  - $Q = \texttt{nil}$. Then by rule (PAR-OP$_1$) 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.
  - $Q \neq \texttt{nil}$. Then $\langle \Gamma, S_1, E_1, Q \upharpoonleft R\rangle \rightarrow \langle \Gamma_1, S_1', E_1', Q' \upharpoonleft R\rangle$ is derived by (PAR-OP$_2$) from $\langle \Gamma, S_1, E_1, Q\rangle \rightarrow \langle \Gamma_1, S_1', E_1', Q'\rangle$. There are two subcases:
    i) Q is $\mathcal{L}$-guarded. Then by Theorem 3.5 $\langle \Gamma, S_2, E_2, Q\rangle \rightarrow \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$, whence by (PAR-OP$_2$) we may conclude.
    ii) Q is not $\mathcal{L}$-guarded. By induction either Q is syntactically high or $\langle \Gamma, S_2, E_2, Q\rangle \rightarrow \langle \Gamma', S_2', E_2', Q'\rangle$ with $\langle S_1', E_1'\rangle =_{\mathcal{L}}^{\Gamma'} \langle S_2', E_2'\rangle$, in which case we conclude using (PAR-OP$_2$). In the former 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 Lemma 3.4 R is syntactically high, and thus by Definition 3.4 also $P = Q \upharpoonleft R$ is syntactically high.

  Suppose now $\langle E_1, Q\rangle\ddagger$. In this case it must be $\neg\langle E_1, R\rangle\ddagger$ and by (PAR-OP$_3$) $\langle \Gamma, S_1, E_1, Q \upharpoonleft R\rangle \rightarrow \langle \Gamma, S_1, E_1, R \upharpoonleft Q\rangle$. Here there are four possibilities:

  - Q and R are both non $\mathcal{L}$-guarded. Then, as in the corresponding case of *Proof of (1)*, we deduce that both Q and R are syntactically high and thus also $P = Q \upharpoonleft R$ is syntactically high.
  - Q is $\mathcal{L}$-guarded and R is not. In this case we may use Theorem 3.5 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-OP$_3$) 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 $P = Q \upharpoonleft R$ is syntactically high.
  - R is $\mathcal{L}$-guarded and Q is not. By Theorem 3.5 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-OP$_3$) 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 $P = Q \upharpoonleft R$ is syntactically high.
  - Q and R are both $\mathcal{L}$-guarded. In this case by Theorem 3.5 we have $\langle E_2, Q\rangle\ddagger$ and $\neg\langle E_2, R\rangle\ddagger$, and we conclude immediately using rule (PAR-OP$_3$).

- (SUB) Here $\Gamma \vdash P : (\theta, \sigma)\ cmd$ is deduced from $\Gamma \vdash P : (\theta', \sigma')\ cmd$ for some θ', σ' such that $\theta' \geq \theta$ and $\sigma' \leq \sigma$. We may then conclude immediately, using Theorem 3.5 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 (first clause of the forthcoming Definition 3.6) instead of requiring them to preserve low-equality of memories, as required for "low" programs (second and third clause of Definition 3.6). This can be explained as follows. Recall that in reactive computations all signals are reset to absent at the beginning of an instant. This means that the low signal environment is not, in general, preserved by instant changes. it is empty. As a consequence, two semantically high programs resulting from a fork after a high test may have different effects on the low signal environment, since one of them may jump to the next instant while the other one does not. Since this difference does not arise from low assignments or low signal emissions, but only from the passage of instants, it seems reasonable to abstract from it. On the other hand, the passage of instants will be observable for "low" programs.

As for the low store, it will have to be preserved both by low and high programs: in case of semantically high programs, this is implied by the first clause of our bisimulation. Indeed, a semantically high program preserves the low store by definition (*cf* Def. 3.4). So, if the two compared programs are semantically high, when run in low-equal stores they will again produce low-equal stores. In fact, in the next section we will show that our reactive notion of security implies a more standard one, which ignores signals and only requires the low store to be preserved.

Definition 3.6 (Reactive $\mathcal{L}$-bisimulation) *Let $\mathcal{L}$ be a downward-closed set of security levels. 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 that $\langle S_1, E_1\rangle =_{\mathcal{L}}^{\Gamma_1\cap\Gamma_2} \langle S_2, E_2\rangle$ and that one of the following properties holds, where $C_i' = \langle \Gamma_i', S_i', E_i', P_i'\rangle$:*

(1) $P_i \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_i,\mathcal{L}}$ *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) $\langle \Gamma_1, S_1, E_1, P_1\rangle \to \langle \Gamma_1', S_1', E_1', P_1'\rangle$ *with* $(n \in \mathtt{dom}(\Gamma_1'\backslash\Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2))$ *implies* $\langle \Gamma_2, S_2, E_2, P_2\rangle \to \langle \Gamma_2', S_2', E_2', P_2'\rangle$ *with* $(n \in \mathtt{dom}(\Gamma_2'\backslash\Gamma_2) \Rightarrow n \in \mathtt{dom}(\Gamma_1'\backslash\Gamma_1))$ *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). 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 [6].

Let us illustrate more precisely the use of these conditions with a couple of examples. Consider the program $P = (\texttt{let}\ x : L = 0\ \texttt{in}\ y_L := x)$. Let Γ_1 and Γ_2 be the type environments defined by $\Gamma_1 : y_L \mapsto L\,,\ \Gamma_2 : x \mapsto L,\ y_L \mapsto L\,.$ Note that P is not semantically high in Γ_1, Γ_2. 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 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 $x \in \texttt{dom}(\Gamma_2)$, 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 to x. Note that the pair of configurations (C_1, C_2) is reachable [7] by running the program $Q = (\texttt{if}\ z_H = 0\ \texttt{then}\ (\texttt{nil}; P)\ \texttt{else}\ (\texttt{let}\ x : L = 1\ \texttt{in}\ P))$ in two identical type 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 $P_1 = (\texttt{let}\ x : L = 0\ \texttt{in}\ x := x + 1)$ and $P_2 = (\texttt{let}\ x : L = 1\ \texttt{in}\ x := x + 1)$, 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 position 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

[6] In fact, these two conditions on new names are not necessary for our soundness result, but they make sense for arbitrary configurations and they render our security notion stronger, as will be made clear in the following discussion.

[7] up to the addition of the global variable z_H to both the Γ_i's and the S_i's.

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 reachable [6], by running the (not semantically high) program $Q = (\texttt{if } z_H = 0 \texttt{ then } P_1 \texttt{ else } P_2)$ 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 $\bar{S}_2(z_H) \neq 0$. Hence the second condition is somehow dual to the first, in that it ensures that configurations are not equated by accident.

The set of *secure programs* is now defined, as usual, 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 3.7 (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 $\langle S_1, E_1\rangle =^{\Gamma}_{\mathcal{L}} \langle S_2, E_2\rangle$, 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, when comparing a program with itself in Definition 3.7, we require it to be the same in the two initial configurations, it will turn out that the two conditions of Clause (3) in Definition 3.6 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 may always evolve, by Theorems 3.5 and 3.6 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, where the executing program is not necessarily typable, as shown by the examples above. Moreover the first one will be necessary for proving Theorem 3.8, which compares our bisimulation with a more standard one.

3.4 Soundness of the type system

In this section we establish our soundness result, i.e. we prove that every typable program is secure. This result rests heavily on the Theorems 3.5 and 3.6 proved in the previous section, which describe the one-step behaviour of typable programs (respectively low-guarded and non low-guarded). We also introduce another notion of bisimulation, and prove that it is weaker than reactive bisimulation.

Theorem 3.7 (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 $i = 1, 2$, $\langle S_1, E_1 \rangle =_\mathcal{L}^{\Gamma_1 \cap \Gamma_2} \langle S_2, E_2 \rangle$ and one of the following holds:

(1) $P_i \in \mathcal{H}_{\mathtt{syn}}^{\Gamma_i, \mathcal{L}}$ 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 by Clause (2) $\langle \Gamma, S_1, E_1, P \rangle \ \mathcal{S}_\mathcal{L} \ \langle \Gamma, S_2, E_2, P \rangle$. 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:

- $P_i \in \mathcal{H}_{\mathtt{syn}}^{\Gamma_i, \mathcal{L}}$ for $i = 1, 2$, by Clause (1). Since $\mathcal{H}_{\mathtt{syn}}^{\Gamma_i, \mathcal{L}} \subseteq \mathcal{H}_{\mathtt{sem}}^{\Gamma_i, \mathcal{L}}$ we have then $P_i \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_i, \mathcal{L}}$ and therefore $C_1 \sim_\mathcal{L} C_2$ by Clause (1) of Def. 3.6.
- $\langle \Gamma_1, P_1 \rangle = \langle \Gamma_2, P_2 \rangle$, by Clause (2). We may assume $P_i \notin \mathcal{H}_{\mathtt{syn}}^{\Gamma_i, \mathcal{L}}$, 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 3.5 or Theorem 3.6 depending on whether P_1 is low-guarded or not (in the latter case we also use the fact that $P_1 \notin \mathcal{H}_{\mathtt{syn}}^{\Gamma_1, \mathcal{L}}$) we may deduce $C_2 \hookrightarrow C_2'$ (resp., $C_2 \rightarrow C_2'$), 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 $\langle S_1', E_1' \rangle =_\mathcal{L}^{\Gamma_1' \cap \Gamma_2'} \langle S_2', E_2' \rangle$. In the case where $C_1 \rightarrow C_1'$ is matched by $C_2 \rightarrow C_2'$, the condition $(n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2))$ is satisfied because $\Gamma_2 = \Gamma_1$, and the condition $(n \in \mathtt{dom}(\Gamma_2' \backslash \Gamma_2) \Rightarrow n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1))$ 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 standard kind of bisimulation where programs are only required to preserve the low store and semantically 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 3.8 (Weak reactive $\mathcal{L}$-Bisimulation) *Let $\mathcal{L}$ be a downward-closed set of security levels. 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 that $S_1 =_{\mathcal{L}}^{\Gamma_1 \cap \Gamma_2} S_2$ and that the following property holds:*

$\langle \Gamma_1, S_1, E_1, P_1\rangle \longmapsto \langle \Gamma_1', S_1', E_1', P_1'\rangle$ *with* $(n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2))$ *implies*

$\langle \Gamma_2, S_2, E_2, P_2\rangle \longmapsto^{\star} \langle \Gamma_2', S_2', E_2', P_2'\rangle$ *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 3.6 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 3.8 by the possibility that a move be simulated by the empty move. To see this, let us look back at the first pair of programs P_1 and P_2 defined at page 49. Clearly, the second condition of Definition 3.6 would not help us distinguish $C_1 = \langle \emptyset, \emptyset, \emptyset, P_1\rangle$ 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 we have that $C_1 \simeq_{\mathcal{L}} C_2$, 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 second example (C_1, C_2) used for $\sim_{\mathcal{L}}$ at page 49, 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 $x \in \mathtt{dom}(\Gamma_2)$, nor by idling since $S_1'(x) \neq S_2(x)$.

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

Definition 3.9 (Γ- Weakly Secure Programs) *P is weakly secure in Γ if for any downward-closed $\mathcal{L} \subseteq \mathcal{T}$ 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 in weak reactive bisimulation $\simeq_{\mathcal{L}}$. To see that $\simeq_{\mathcal{L}} \not\subseteq \sim_{\mathcal{L}}$ consider the program $P = (\mathtt{if}\ x_H = 0\ \mathtt{then}\ \mathtt{emit}\ a_L\ \mathtt{else}\ \mathtt{emit}\ b_L)$. Clearly, if Γ is the type 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 3.8 ($\sim_{\mathcal{L}}$ 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 $\mathcal{R}_{\mathcal{H}}$ on configurations as follows:

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

We show that the relation $\mathcal{R} = \sim_{\mathcal{L}} \cup \mathcal{R}_{\mathcal{H}}$ is a weak reactive bisimulation. Let $C_i = \langle \Gamma_i, S_i, E_i, P_i \rangle$.

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 $C_1 \longmapsto C_1' = \langle \Gamma_1', S_1', E_1', P_1' \rangle$, with $n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2)$. There are two cases to consider:

- $P_i \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_i,\mathcal{L}}$ 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 3.4 we have $P_1' \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_1',\mathcal{L}}$, with $\langle S_1, E_1 \rangle =_{\mathcal{L}}^{\Gamma_1} \langle S_1', E_1' \rangle$. Correspondingly we can choose $C_2 \rightarrow^{\star} C_2' = \langle \Gamma_2', S_2', E_2', P_2' \rangle$, 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 $P_2' \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_2,\mathcal{L}}$ and what is left to show is that $S_1' =_{\mathcal{L}}^{\Gamma_1' \cap \Gamma_2} S_2$. Since $n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2)$, we have $\Gamma_1' \cap \Gamma_2 = \Gamma_1 \cap \Gamma_2$. Moreover, since $x \in \mathtt{dom}(S_1' \backslash S_1)$ implies $x \notin \mathtt{dom}(\Gamma_1)$ (because of the condition in the operational rule (LET)) and a fortiori $x \notin \mathtt{dom}(\Gamma_1 \cap \Gamma_2)$, 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 3.4 we have $P_1' \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_1',\mathcal{L}}$. 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 $C_2 \longmapsto^{\star} C_2$. Then, as in the previous case, $P_2' \in \mathcal{H}_{\mathtt{sem}}^{\Gamma_2,\mathcal{L}}$ 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 $C_1' \ \mathcal{R}_{\mathcal{H}} \ C_2'$.
- $P_i \notin \mathcal{H}_{\mathtt{sem}}^{\Gamma_i,\mathcal{L}}$ for some $i \in \{1, 2\}$. Then, if the transition is of the form $C_1 \hookrightarrow C_1'$ we know by Clause (2) of Definition 3.6 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 $n \in \mathtt{dom}(\Gamma_1' \backslash \Gamma_1) \Rightarrow n \notin \mathtt{dom}(\Gamma_2)$, we know by Clause (3) of Def. 3.6 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.

□

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 this is not the case. Consider for instance the program $P = \texttt{if}\ x_H = 0\ \texttt{then}\,(\texttt{let}\ u : L = 0\ \texttt{in}\ u := 0)\ \texttt{else}\,(\texttt{let}\ v : L = 1\ \texttt{in}\ v := 1)$. Program P 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, the program Q defined by $Q = \texttt{if}\ x_H = 0\ \texttt{then}\ (\texttt{let}\ u : L = 0\ \texttt{in}\ z_L := u)\ \texttt{else}\ (\texttt{let}\ v : L = 1\ \texttt{in}\ z_L := v)$ is neither secure nor weakly secure, since it may assign different values to the low global variable z_L.

Indeed, there are at least three reasons why $\sim_{\mathcal{L}}$ is stronger than $\simeq_{\mathcal{L}}$.

The first reason 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 above and by the first example at page 52.

The third reason has to do with the difference in granularity between the two bisimulations, namely with the fact that reactive bisimulation compares two configurations whose programs are not both semantically high by running them in lockstep (one step must be simulated by exactly one step), while weak reactive bisimulation allows one step of one configuration to be always simulated by a sequence of steps of the other. Consider the program $P = \texttt{if}\ x_H = 0\ \texttt{then loop}\ (y_L := 0\,;\ y_L := 1)\ \texttt{else loop}\ (y_L := 1\,;\ y_L := 0)$. Let us call P_1 and P_2 the two branches of the conditional, and suppose P runs in the type environment $\Gamma : y_L \mapsto L, x_H \mapsto H$ and in any pair of low-equal stores S_1, S_2 such that $S_1(x_H) = 0$, $S_2(x_H) \neq 0$ and $S_i(y_L) \notin \{0, 1\}$. Then, P will evolve to P_1 when run in the first memory and to P_2 when run in the second. If now P_1 moves to P_1', then the value of y_L will be changed to 0. This move cannot be simulated by a single move of P_2, while it can be simulated by a sequence of two moves. Hence P is weakly secure but not secure in Γ.

4 Conclusion

In this paper we have addressed the question of noninterference for reactive programs. We have presented a type system guaranteeing noninterference in a simple imperative reactive language. We aim to extend our results to a fully-fledged call-by-value language for mobility built around a reactive core, called ULM [11], which is currently under study. For this purpose, we intend

to put together the results and techniques developed in the present paper with the work of [4] on the non-disclosure policy for the functional kernel of ULM (extended with thread creation and a declassification construct), and with the work on non-disclosure for mobile programs carried out in [2,3].

We have studied information flow for a concurrent language involving explicit synchronization primitives, showing that new kinds of leaks arise from synchronization. Another paper that examines the impact of synchronization on information flow is [20]. However the analogy between [20] and our work cannot be pushed very far, since [20] does not consider explicitly the scheduling or a notion of instant, relying instead on asynchronous parallel composition.

As has been observed, reactive programs obey a fixed scheduling policy, which is enforced here in a syntactic way by means of the synchronous parallel construct $\upharpoonleft$. Other approaches to the study of noninterference in the presence of scheduling include the probabilistic one, proposed for instance in [26] and [22]. 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 [14], which allows different scheduling policies to be expressed, and [22], 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, as well as proposing a type system to ensure it, according to a now classical methodology.

From this starting point, a few improvements may be envisaged, concerning both the scheduling policy and the type system. We are currently investigating a different scheduling policy for the purely reactive language of [6], which records the order of creation of threads in such a way that the thread pointer of the scheduler can be reset to the first (live) created thread at the beginning of each instant. We hope in this way to be able to lift the constraint for typing synchronous parallel composition, which is at the moment quite restrictive and makes our type system somehow "non compositional" (in the sense that composing two typable programs in parallel does not always result in a typable program). As concerns the type system, it would be worth studying a refinement along the lines of the recent work [12]. Indeed, as noted in Section 3, with cooperative scheduling some typical leaks arising with nondeterministic scheduling can be avoided, like those originated by high conditionals with terminating branches of different lengths. We could then relax the typing rule for conditionals by stipulating that they contribute to the guard level only if one of their branches does not terminate (a sufficient condition for termination being the absence of loops and `when` commands).

Another issue that deserves more investigation is the security notion. We have examined two notions here, incorporating a different observation of the allo-

cation of local names. According to our first notion, reactive bisimulation, the name allocator itself needs to be "secure", in the sense that the same sequence of low-observable names must be generated in low-equivalent runs of a secure program. The second notion, weak reactive bisimulation, implements a laxer observation of local names. One can imagine further variations on these notions, where the observation of local names is only allowed indirectly, that is, when they are used to implement an illegal flow towards a global name. A final point to note about our notion of security is that, given the determinacy of reactive computations, it could as well be defined as a trace equivalence rather than as a bisimulation. The trace-based definition can be easily derived from the bisimulation-based one, in a way similar to that used in [14] for a different language. It may be worth noting here that this trace-equivalence would coincide with reactive bisimulation and not with the weak reactive one. Indeed, as shown by the last example at page 54, weak reactive bisimulation does not respect execution traces.

Acknowledgments

We would like to thank David Sands for his suggestion of proving a refinement result for 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. Many thanks also to Frédéric Boussinot for discussions about reactive programming and for suggesting the service-replacement example. Finally, we wish to thank the anonymous referees for insightful comments and helpful feedback.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings POPL'00*, pages 40–53, 2000.

[2] A. Almeida Matos. Non-disclosure for distributed mobile code. In *Proceedings FST-TCS'05*, volume 3821 of *Lecture Notes in Computer Science*, pages 177–188, 2005.

[3] A. Almeida Matos. *Typing secure information flow: declassification and mobility.* PhD thesis, École des Mines de Paris, 2006.

[4] A. Almeida Matos and G. Boudol. On declassification and the non-disclosure policy. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 226–240, 2005.

[5] A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'04)*, 2004. TUCS Research Report n. 31, 2004.

[6] R. Amadio, G. Boudol, F Boussinot, and I. Castellani. Reactive concurrent programming revisited. In *Proceedings of the Workshop "Algebraic process calculi: the first 25 years and beyond"*, 2005. Revised version to appear in ENTCS.

[7] R. Amadio and F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In *Proceedings EXPRESS'05*, 2005. Revised version in ENTCS 154, Issue 3.

[8] R. Amadio and S. Dal Zilio. Resource control for synchronous cooperative threads. In *Proceedings CONCUR'04*, volume 3170 of *Lecture Notes in Computer Science*, 2004.

[9] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, Ph. Le Guernic, and R. de Simone. The synchronous languages twelve years later. In *Proceedings of the IEEE, Special Issue on the Modeling and Design of Embedded Software*, volume 91(1), pages 64–83, 2003.

[10] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comput. Programming*, 19:87–152, 1992.

[11] G. Boudol. ULM, a core programming model for global computing. In *Proceedings ESOP'04*, volume 2986 of *Lecture Notes in Computer Science*, pages 234–248, 2004.

[12] G. Boudol. On typing information flow. In *International Colloquium on Theoretical Aspects of Computing*, volume 3722 of *Lecture Notes in Computer Science*, pages 366–380, 2005.

[13] G. Boudol and I. Castellani. Noninterference for concurrent programs. In *Proceedings ICALP'01*, volume 2076 of *Lecture Notes in Computer Science*, pages 382–395, 2001.

[14] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[15] F. Boussinot. Reactive programming (Software). URL: http://www-sop.inria.fr/mimosa/rp/.

[16] F. Boussinot. Loft+Cyclone. INRIA Research Report 5680, INRIA Sophia Antipolis, 2005.

[17] F. Boussinot and J.-F. Susini. The sugarcubes tool box: a reactive JAVA framework. *Software Practice and Experience*, 28(14):1531–1550, 1998.

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

[19] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings POPL'02*, pages 81–92, 2002.

[20] A. Sabelfeld. The impact of synchronization on secure information flow in concurrent programs. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *Lecture Notes in Computer Science*, pages 227–241, 2001.

[21] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[22] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings 13th IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 200–214, 2000.

[23] G. Smith. A new type system for secure information flow. In *Proceedings 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 115–125, 2001.

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

[25] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop (CSFW'97)*, pages 156–168, 1997.

[26] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2-3):231–253, 1999.

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

[28] S. Zdancewic and A. Myers. Observational determinism for concurrent program security. In *Proceedings 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 29–43, 2003.