Fine-grained and coarse-grained reactive noninterference

Pejman Attar and Ilaria Castellani*

INRIA, 2004 route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex
pejman.attar@inria.fr, ilaria.castellani@inria.fr

Abstract. We study bisimilarity and the security property of *noninterference* in a core *synchronous reactive language* that we name *CRL*.
In the synchronous reactive paradigm, programs communicate by means of broadcast events, and their parallel execution is regulated by the notion of *instant*. Within each instant, programs may emit events and get suspended while waiting for events emitted by other programs. They may also explicitly return the control to the scheduler, thereby suspending themselves until the end of the instant. An instant is thus a period of time during which all programs compute until termination or suspension.
In *CRL* there is no memory, and the focus is on the control structure of programs. An *asymmetric parallel operator* is used to implement a deterministic scheduling. This scheduling is fair – in the sense that it gives its turn to each parallel component – if all components are *cooperative*, namely if they always return the control after a finite number of steps.
We first prove that *CRL* programs are indeed cooperative. This result is based on two features of the language: the semantics of loops, which requires them to yield the control at each iteration of their body; and a delayed reaction to the absence of events, which ensures the monotonicity of computations (viewed as I/O functions on event sets) during instants.
Cooperativeness is crucial as it entails the *reactivity* of a program to its context, namely its capacity to input events from the context at the start of instants, and to output events to the context at the end of instants.
We define two bisimulation equivalences on programs, formalising respectively a *fine-grained observation* of programs (the observer is viewed as a program) and a *coarse-grained observation* (the observer is viewed as part of the context). As expected, the latter equivalence is more abstract than the former, as it only compares the I/O behaviours of programs at each instant, while the former also compares their intermediate results.
Based on these bisimulations, two properties of *reactive noninterference* (RNI) are proposed. Both properties are time-insensitive and termination-insensitive. Coarse-grained RNI is more abstract than fine-grained RNI, because it views the parallel operator as commutative and abstracts away from repeated emissions of the same event during an instant.
Finally, a type system guaranteeing both security properties is presented. Thanks partly to a design choice of *CRL*, which offers two separate constructs for loops and iteration, this type system allows for a precise treatment of termination leaks, which are an issue in parallel languages.

* Work partially supported by the french ANR 08-EMER-010 grant PARTOUT.

1 Introduction

Many systems of widespread use, such as web browsers and web applications, may be modelled as *reactive programs*, that is programs that listen and react to their environment in a continuous way, by means of events. Since the environment may include mutually distrusting parties, such as a local user and a remote web server, reactive programs should be able to protect the confidentiality of the data they manipulate, by ensuring a *secure information flow* from the inputs they receive from one party to the outputs they release to another party.

Secure information flow is often formalised via the notion of *noninterference* (NI), expressing the absence of dependency between secret inputs and public outputs (or more generally, between inputs of some confidentiality level to outputs of lower or incomparable level). Originally introduced in [10], NI has been studied for a variety of languages, ranging from standard imperative and functional languages [14,12] to process calculi based on CCS or the pi-calculus [9]. On the other hand, little attention has been paid to noninterference for reactive programs, with the notable exception of [11], [2] and [5].

We shall focus here on a particular brand of reactive programming, namely the *synchronous* one, which was first embodied in the synchronous language *SL* [7], an offspring of Esterel [4], and later incorporated into various programming environments, such as C, Java, Caml and Scheme. In the synchronous paradigm, the parallel execution of programs is regulated by a notion of *instant*. The model of *SL* departs from that of Esterel in that it assumes the reaction to the absence of an event to be postponed until the end of the instant. This assumption helps disambiguating programs and simplifying the implementation of the language. It is also essential to ensure the monotonicity of programs and their reactivity to the environment.

In this work, we will not explicitly model the interaction of a reactive program with the environment (this could be easily done but it would not bring any further insight). Instead, we concentrate on the interaction *within* a reactive program, making sure it regularly converges to a stable state (end of instant), in which the program is ready to interact with the environment. We call this property *cooperativeness* [1] or *internal reactivity*. In the sequel, we shall abandon the distinction between internal reactivity (among the components of a program) and *external reactivity* (towards the environment), to focus on the former.

This paper attempts to explore "secure reactive programming in a nutshell". To this end, we concentrate on a minimal reactive language without memory, consisting of standard sequential operators, an asymmetric parallel operator $\nmid$ (formalising a kind of *coroutine* parallelism under a deterministic scheduling), together with four typical reactive constructs, which we briefly describe next.

In our *Core Reactive Language CRL*, programs are made of parallel components s, s' – also called "threads" for simplicity in the following – combined with the operator $s \nmid s'$ and communicating by means of broadcast events. Threads may emit events, via a `generate` ev instruction, and get suspended while waiting for events to be emitted by other threads, through an `await` ev instruction. They may also explicitly yield the control to the scheduler, via a `cooperate` instruction, thereby suspending themselves until the end of the current instant. An instant is therefore a period of time during which all threads compute until termination or suspension. Clearly, this is a logical rather than a physical notion of instant, since the termination of instants is determined by the collective behaviour of threads rather than by some physical clock. At the end of an instant, all threads are inactive and share the same view of emitted events. At instant change, a preemption construct `do` s `watching` ev allows some suspended parts of threads to be pruned off, thus implementing a time-out mechanism. Interaction with the environment is limited to the start and the end of instants: the environment injects events at the start of instants and collects them at the end.

The starting point of our work is the paper [2], which laid the basis for the study of noninterference in a synchronous reactive language. The present work improves on [2] in several respects, which we summarise below.

The language examined in [2] is similar to CRL but strictly more expressive, including imperative constructs, local declarations and a memory. Indeed, our asymmetric parallel operator $\nmid$ is inspired by that of [2]. Here, however, we adopt a slightly different semantics for $s \nmid s'$, prescribing a *late cooperation* on the left (s executes up to termination or suspension before giving the control to s') and an *early cooperation* on the right (when getting the control from s, s' executes only until unblocking s - by generating the event that s is waiting for - or, if this is not possible, until termination or suspension). This simple change forces the scheduler to serve the same thread at the start of each instant, thus avoiding the so-called *scheduling leaks* of [2], and allowing for a more relaxed typing rule for $\nmid$, which is just the standard rule for symmetric parallel composition.

Moreover, reactivity was not a concern in [2]: as soon as they contained *while loops*, programs were not guaranteed to terminate or suspend within an instant. Hence, it only made sense to consider a fine-grained notion of noninterference. By contrast, in CRL all programs are reactive, thanks to a clear separation between the loop construct `loop` s and the iteration construct `repeat` exp `do` s, and to our semantics for loops, which requires them to yield the control at each iteration of their body. This makes it possible to define a notion of coarse-grained *reactive noninterference* (RNI), which accounts only for the I/O behaviour of programs within each instant. The coarse-grained RNI property has an advantage over the fine-grained one: it exploits in a more direct way the structure of reactive computations, and it recovers the flavour of big-step semantics within each instant, offering a more abstract NI notion for reactive programs.

Finally, our type system is more permissive than that of [2], thanks to the relaxed typing rule for parallel composition and to refined typing rules for the conditional. Both improvements are made possible by design choices of CRL.

The main contributions of this paper are: 1) the reactivity result, 2) the definition of two bisimulation equivalences for synchronous reactive programs, of different granularity. To our knowledge, semantic equivalences for reactive programs have only been studied previously by Amadio [3]; 3) the proposal of two properties of reactive noninterference, based on the above bisimulations, and 4) the presentation of a type system ensuring both noninterference properties.

The rest of the paper is organised as follows. Sections 2 and 3 present the syntax and the semantics of the language CRL. Section 4 is devoted to proving reactivity of CRL programs. Section 5 introduces the two bisimulation equivalences and gives some properties of them. In Section 6 we define our two NI properties. Section 7 presents our security type system and the proof of its soundness. Finally, future and related work are briefly discussed in Section 8.

2 Syntax

In this section we introduce the syntax of CRL. Let Val be a set of values, ranged over by v, v', Var a set of variables, ranged over by x, y, z, and $Events$ a set of events, ranged over by ev, ev'. A fixed valuation function $V : Var \rightarrow Val$ for open terms is assumed, which however will be left implicit until Section 6.

Expressions. An expression $exp \in Exp$ may be a basic value, a variable, or the value returned by a function. Letting $\overrightarrow{exp}$ denote a tuple of expressions $exp_1, \ldots, exp_n$, the syntax of expressions is:

$$exp \in Exp ::= v \mid x \mid f(\overrightarrow{exp})$$

The evaluation of a function call $f(\overrightarrow{exp})$ is assumed to be instantaneous, and therefore so is the evaluation of an expression exp, denoted by $exp \rightsquigarrow v$, which is formally defined by the three rules:

$$\frac{}{v \rightsquigarrow v} \qquad \frac{V(x) = v}{x \rightsquigarrow v} \qquad \frac{\forall i \in \{1, \ldots, n\} \,.\, exp_i \rightsquigarrow v_i \qquad f(v_1, \ldots, v_n) = v}{f(\overrightarrow{exp}) \rightsquigarrow v}$$

Programs. We now present the syntax of CRL programs. Alongside with typical sequential operators, CRL includes four operators that are commonly found in reactive languages, `cooperate`, `generate` ev, `await` ev and `do` s `watching` ev, as well as a binary *asymmetric parallel operator*, denoted by $\dagger$, which performs a deterministic scheduling on its components. This operator is very close to that used in [2] and, earlier on, in the implementation of *SugarCubes* [8]. However, while in [2] and [8] each parallel component was executing as long as possible, our operator $\dagger$ implements a form of *prioritised scheduling*, where the first component yields the control only when terminating or suspending (*late cooperation*), while the second yields it as soon as it generates an event that unblocks the first component (*early cooperation*).

$$\begin{aligned} s \in Programs ::= {} & \texttt{nothing} \mid s; s \mid (s \dagger s) \mid \\ & \texttt{cooperate} \mid \texttt{generate}\, ev \mid \texttt{await}\, ev \mid \texttt{do}\, s\, \texttt{watching}\, ev \mid \\ & (\texttt{loop}\, s) \mid (\texttt{repeat}\, exp\, \texttt{do}\, s) \mid (\texttt{if}\, exp\, \texttt{then}\, s\, \texttt{else}\, s) \end{aligned}$$

Note that our language includes two different constructs for loops and iteration, in replacement of the standard *while loop* operator. This allows for a clear separation between nonterminating behaviours and iterative behaviours.

3 Semantics

This section presents the operational semantics of *CRL*. Programs proceed through a succession of instants, transforming sets of events. There are two transition relations, both defined on *configurations* of the form $\langle s, E \rangle$, where s is a program and $E \subseteq Events$ is an *event environment*, i.e. a set of present events.

Let us first give the general idea of these two transition relations:

1. The *small-step transition relation* describes the step-by-step execution of a configuration within a an instant. The general format of a transition is:

   $$\langle s, E \rangle \rightarrow \langle s', E' \rangle$$

   where:
   - s is the program to execute and s' is the residual program;
   - E is the starting event environment and E' is the resulting event environment: E' coincides with E if the transition does not generate any new event; otherwise $E' = E \cup \{ev\}$, where ev is the new generated event.
2. The *tick transition relation* describes the passage from one instant to the next, and applies only to suspended configurations. A transition of this kind has always the form:

   $$\langle s, E \rangle \hookrightarrow \langle [s]_E, \emptyset \rangle$$

   where the resulting event environment is empty and $[s]_E$ is a "reconditioning" of program s for the next instant, possibly allowing it to resume execution at the next instant even without the help of new events from the environment.

Before formally defining $\rightarrow$ and $\hookrightarrow$, we introduce the *suspension predicate* $\langle s, E \rangle \ddagger$, which holds when s is suspended in the event environment E, namely when s waits for some event not contained in E, or when s deliberately yields the control for the current instant by means of a `cooperate` instruction.

The rules defining the predicate $\ddagger$ and the relations $\rightarrow$ and $\hookrightarrow$ are given in Fig. 3. The *reconditioning function* $[s]_E$ prepares s for the next instant: it erases all guarding `cooperate` instructions, as well as all guarding `do` s' `watching` ev instructions whose time-out event ev is in E (i.e. has been generated).

Let us comment on the most interesting transition rules. The execution of a parallel program always starts with its left branch (Rule (par_1)). Once the left branch is over, the program reduces to its right branch (Rule (par_2)). If the left branch is suspended, then the right branch executes (Rule (par_3)) until unblocking the left branch. Thus *early cooperation* is required in the right branch. To avoid nondeterminism, a terminated right branch can only be eliminated if the left branch is suspended (Rule (par_4)). A `loop` s program executes its body cyclically: a `cooperate` instruction is systematically added in parallel to its body to avoid *instantaneous loops*, i.e. divergence within an instant[1] (Rule ($loop$)). A `do` s `watching` ev program executes its body until termination or suspension (Rule ($watch_1$)), reducing to `nothing` when its body terminates (Rule ($watch_2$)), and getting processed by the reconditioning function when its body suspends.

The small-step transition relation satisfies two simple properties: determinism and incremental production of events throughout an instant.

[1] In general, we shall call "instantaneous" any property that holds within an instant.

$$\langle cooperate, E\rangle\ddagger \quad (coop) \qquad \frac{ev \notin E}{\langle \texttt{await}\ ev, E\rangle\ddagger} \quad (wait_s) \qquad \frac{\langle s, E\rangle\ddagger}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, E\rangle\ddagger} \quad (watch_s)$$

$$\frac{\langle s_1, E\rangle\ddagger}{\langle s_1; s_2, E\rangle\ddagger} \quad (seq_s) \qquad \frac{\langle s_1, E\rangle\ddagger \qquad \langle s_2, E\rangle\ddagger}{\langle s_1 \dagger s_2, E\rangle\ddagger} \quad (par_s) \qquad \frac{\langle s, E\rangle\ddagger}{\langle s, E\rangle \hookrightarrow \langle [s]_E, \emptyset\rangle} \quad (tick)$$

Suspension Predicate and Tick Transition Rule

$$[\texttt{cooperate}]_E = \texttt{nothing} \qquad [\texttt{do}\ s\ \texttt{watching}\ ev]_E = \begin{cases} \texttt{nothing} & if \quad ev \in E \\ \texttt{do}\ [s]_E\ \texttt{watching}\ ev & otherwise \end{cases}$$

$$[\texttt{await}\ ev]_E = \texttt{await}\ ev \qquad [s_1; s_2]_E = [s_1]_E\ ; s_2 \qquad [s_1 \dagger s_2]_E = [s_1]_E \dagger [s_2]_E$$

Reconditioning Function

$$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1; s_2, E\rangle \to \langle s_1'; s_2, E'\rangle} \quad (seq_1) \qquad \langle \texttt{nothing}\ ; s, E\rangle \to \langle s, E\rangle \quad (seq_2)$$

$$\frac{\langle s_1, E\rangle \to \langle s_1', E'\rangle}{\langle s_1 \dagger s_2, E\rangle \to \langle s_1' \dagger s_2, E'\rangle} \quad (par_1) \qquad \langle \texttt{nothing} \dagger s, E\rangle \to \langle s, E\rangle \quad (par_2)$$

$$\frac{\langle s_1, E\rangle\ddagger \qquad \langle s_2, E\rangle \to \langle s_2', E'\rangle}{\langle s_1 \dagger s_2, E\rangle \to \langle s_1 \dagger s_2', E'\rangle} \quad (par_3) \qquad \frac{\langle s, E\rangle\ddagger}{\langle s \dagger \texttt{nothing}, E\rangle \to \langle s, E\rangle} \quad (par_4)$$

$$\langle \texttt{generate}\ ev, E\rangle \to \langle \texttt{nothing}, E \cup \{ev\}\rangle \ (gen) \qquad \frac{ev \in E}{\langle \texttt{await}\ ev, E\rangle \to \langle \texttt{nothing}, E\rangle} \ (wait)$$

$$\frac{\langle s, E\rangle \to \langle s', E'\rangle}{\langle \texttt{do}\ s\ \texttt{watching}\ ev, E\rangle \to \langle \texttt{do}\ s'\ \texttt{watching}\ ev, E'\rangle} \quad (watch_1)$$

$$\langle \texttt{do}\ \texttt{nothing}\ \texttt{watching}\ ev, E\rangle \to \langle \texttt{nothing}, E\rangle \quad (watch_2)$$

$$\langle \texttt{loop}\ s, E\rangle \to \langle (s \dagger \texttt{cooperate}); \texttt{loop}\ s, E\rangle \quad (loop)$$

$$\frac{exp \rightsquigarrow n \qquad n \geq 1}{\langle \texttt{repeat}\ exp\ \texttt{do}\ s, E\rangle \to \langle \underbrace{s; \ldots; s}_{n\ times}, E\rangle} \quad (repeat)$$

$$\frac{exp \rightsquigarrow tt}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E\rangle \to \langle s_1, E\rangle} \quad (if_1) \qquad \frac{exp \rightsquigarrow ff}{\langle \texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2, E\rangle \to \langle s_2, E\rangle} \quad (if_2)$$

Small-step Transition Rules

Fig. 1. Operational Semantics of *CRL*

Proposition 1. (Determinism)

Let $s \in Programs$ and $E \subseteq Events$. Then:

$$s \neq \texttt{nothing} \quad \Rightarrow \quad \textit{either } \langle s, E\rangle \ddagger \ \textit{ or } \ \exists!\, s', E' \,.\, \langle s, E\rangle \rightarrow \langle s', E'\rangle$$

Proof. By inspecting the suspension and transition rules, it is immediate to see that at most one transition rule applies to each configuration $\langle s, E\rangle$.

Lemma 1. (Event persistence)

$$\textit{Let } s \in Programs \textit{ and } E \subseteq Events. \textit{ Then:} \quad \langle s, E\rangle \rightarrow \langle s', E'\rangle \;\Rightarrow\; E \subseteq E'$$

Proof. Straightforward, since the only transition rule that changes the event environment E is the rule for `generate` ev, which adds the event ev to E.

We define now the notion of *instantaneous convergence*, which is at the basis of the reactivity property of CRL programs. Let us first introduce some notation.

The *timed multi-step transition relation* $\langle s, E\rangle \Rightarrow_n \langle s', E'\rangle$ is defined by:

$$\langle s, E\rangle \Rightarrow_0 \langle s, E\rangle$$

$$\langle s, E\rangle \rightarrow \langle s', E'\rangle \;\wedge\; \langle s', E'\rangle \Rightarrow_n \langle s'', E''\rangle \quad \Rightarrow \quad \langle s, E\rangle \Rightarrow_{n+1} \langle s'', E''\rangle$$

Then the *multi-step transition relation* $\langle s, E\rangle \Rightarrow \langle s', E'\rangle$ is given by:

$$\langle s, E\rangle \Rightarrow \langle s', E'\rangle \;\Leftrightarrow\; \exists\, n \,.\, \langle s, E\rangle \Rightarrow_n \langle s', E'\rangle$$

We define now the relation and the predicate of *instantaneous convergence*:

Definition 1. (Instantaneous convergence)

$$\langle s, E\rangle \Downarrow \langle s', E'\rangle \quad \text{if} \quad \langle s, E\rangle \Rightarrow \langle s', E'\rangle \;\wedge\; (s' = \texttt{nothing} \;\vee\; \langle s', E'\rangle \ddagger)$$

$$\langle s, E\rangle \Downarrow \quad \text{if} \quad \exists s', E' \,.\, \langle s, E\rangle \Downarrow \langle s', E'\rangle$$

The relation and predicate of *instantaneous termination* are defined similarly:

Definition 2. (Instantaneous termination)

$$\langle s, E\rangle \ \Downarrow\!\!\!\!-\ E' \quad \text{if} \quad \langle s, E\rangle \Downarrow \langle \texttt{nothing}, E'\rangle$$

$$\langle s, E\rangle \ \Downarrow\!\!\!\!- \quad \text{if} \quad \exists\, E' \,.\, \langle s, E\rangle \ \Downarrow\!\!\!\!-\ E'$$

The relation $\langle s, E\rangle \Downarrow \langle s', E'\rangle$ defines the overall effect of the program s within an instant, starting with the set of events E. Indeed, $\Downarrow$ may be viewed as defining the *big-step semantics* of programs within an instant[2].

As an immediate corollary of Proposition 1, the relation $\Downarrow$ is deterministic.

The *timed* versions of $\langle s, E\rangle \Downarrow \langle s', E'\rangle$, $\langle s, E\rangle \Downarrow$, $\langle s, E\rangle \ \Downarrow\!\!\!\!-\ \langle s', E'\rangle$ and $\langle s, E\rangle \ \Downarrow\!\!\!\!-$ are defined in the expected way.

In the next section we prove an important property of CRL, namely that every configuration $\langle s, E\rangle$ instantaneously converges. This is true in particular for initial configurations, where $E = \emptyset$. This property is called *reactivity*.

[2] A direct definition of the big-step arrow $\Downarrow$ by a set of structural rules would be slightly more involved, as it would require calculating the output set E' as a fixpoint.

4 Reactivity

In this section we present our first main result, the reactivity of *CRL* programs. In fact, we shall prove a stronger property than reactivity, namely that every configuration $\langle s, E\rangle$ instantaneously converges in a number of steps which is bounded by the *instantaneous size* of s, denoted by $size(s)$. The intuition for $size(s)$ is that the portion of s that sequentially follows a `cooperate` instruction should not be taken into account, as it will not be executed in the current instant. Moreover, if s is a loop, $size(s)$ should cover a single iteration of its body.

To formally define the function $size(s)$, we first introduce an auxiliary function $dsize(s)$ (where"d" stands for "decorated") that assigns to each program an element of $(\mathbf{Nat} \times \mathbf{Bool})$. Then $size(s)$ will be the first projection of $dsize(s)$. Intuitively, if $dsize(s) = (n, b)$, then n is an upper bound for the number of steps that s can execute within an instant; and b is tt or $f\!f$ depending on whether or not a `cooperate` instruction is reached within the instant. For conciseness, we let $n^\wedge$ stand for (n, tt), n stand for $(n, f\!f)$, and n° range over $\{n^\wedge, n\}$.

The difference between $n^\wedge$ and n will essentially show when computing the size of a sequential composition: if the decorated size of the first component has the form $n^\wedge$, then a `cooperate` has been met and the counting will stop; if it has the form n, then n will be added to the decorated size of the second component.

Definition 3. (Instantaneous size)

The function $size : Programs \rightarrow \mathbf{Nat}$ *is defined by:*

$$size(s) = n \quad \textit{if} \quad (dsize(s) = n \ \vee \ dsize(s) = n^\wedge).$$

where the function $dsize : Programs \rightarrow (\mathbf{Nat} \times \mathbf{Bool})$ *is given inductively by:*

$$dsize(\texttt{nothing}) = 0 \qquad\qquad dsize(\texttt{cooperate}) = 0^\wedge$$

$$dsize(\texttt{generate}\ ev) = dsize(\texttt{await}\ ev) = 1$$

$$dsize(s_1; s_2) = \begin{cases} {n_1}^\wedge & \textit{if } dsize(s_1) = {n_1}^\wedge \\ (1 + n_1 + n_2)^\circ & \textit{if } dsize(s_1) = n_1 \ \wedge \ dsize(s_2) = {n_2}^\circ \end{cases}$$

$$dsize(s_1 \dagger s_2) = \begin{cases} (1 + n_1 + n_2)^\wedge & \textit{if } dsize(s_1) = {n_1}^\wedge \ \wedge \ dsize(s_2) = n_2 \\ (1 + n_1 + n_2)^\wedge & \textit{if } dsize(s_1) = n_1 \ \wedge \ dsize(s_2) = {n_2}^\wedge \\ (1 + n_1 + n_2)^\circ & \textit{if } dsize(s_1) = {n_1}^\circ \ \wedge \ dsize(s_2) = {n_2}^\circ \end{cases}$$

$$dsize(\texttt{repeat}\ exp\ \texttt{do}\ s) = (m + (m \times n))^\circ \qquad \textit{if } dsize(s) = n^\circ \ \wedge \ exp \rightsquigarrow m$$

$$dsize(\texttt{loop}\ s) = (2 + n)^\wedge \qquad \textit{if} \quad dsize(s) = n^\circ$$

$$dsize(\texttt{do}\ s\ \texttt{watching}\ ev) = (1 + n)^\circ \quad \textit{if } dsize(s) = n^\circ$$

$$dsize(\texttt{if}\ exp\ \texttt{then}\ s_1\ \texttt{else}\ s_2) = \begin{cases} (1 + max\{n_1, n_2\})^\wedge, \textit{if } dsize(s_i) = {n_i}^\wedge, \\ (1 + max\{n_1, n_2\}), \textit{if for } i \neq j \\ \qquad\qquad dsize(s_i) = n_i \ \wedge \ dsize(s_j) = {n_j}^\circ \end{cases}$$

It may be proven that $size(s)$ decreases along small-step execution:

Lemma 2. (Size reduction within an instant)

$$\forall s\, \forall E \quad (\, \langle s, E \rangle \rightarrow \langle s', E' \rangle \;\Rightarrow\; size(s') < size(s)\,)$$

To prove reactivity, we will use the following Lemma, which establishes that the termination capacities of a program are preserved by event generation, and that feeding a larger event environment in input will produce a larger event environment in output. It is important to notice that the terminating computation will not in general be the same in both cases, since a larger input environment will cause fewer control switches and thus longer turns for each parallel component. However, the two computations will have the same length.

Lemma 3. (Monotonicity of terminating computations)

$$\forall s\, \forall E \quad (\, \langle s, E \rangle \;\Downarrow_n E' \;\Rightarrow\; \forall E_1 \supset E \;\; \exists E'_1 \supseteq E' \;\; \langle s, E_1 \rangle \;\Downarrow_n E'_1\,)$$

We are now ready to prove our main result, namely that every program s instantaneously converges in a number of steps that is bounded by $size(s)$.

Theorem 1. (Script reactivity) $\forall s, \forall E \quad (\, \exists\, n \leq size(s) \;\; \langle s, E \rangle \;\Downarrow_n \,)$

The proof proceeds by simultaneous induction on the structure and on the size of s. Induction on the size will be needed in the case $s = s_1 \nmid s_2$. Lemma 3 will be used in the case $s =$ `repeat` exp `do` s_1. It guarantees that if s_1 terminates in n_1 steps, then each successive iteration of s_1 also terminates in n_1 steps.

5 Fine-grained and coarse-grained bisimilarity

We now introduce two bisimulation equivalences (aka *bisimilarities*) on programs, which differ for the granularity of the underlying notion of observation. The first bisimulation formalises a *fine-grained observation* of programs: the observer is viewed as a program, which is able to interact with the observed program at any point of its execution. The second reflects a *coarse-grained observation* of programs: here the observer is viewed as part of the environment, which interacts with the observed program only at the start and the end of instants.

Let us start with an informal description of the two bisimilarities:

1. *Fine-grained bisimilarity* $\approx^{fg}$. In the bisimulation game, each small step must be simulated by a (possibly empty) sequence of small steps, and each instant change must be simulated either by an instant change, in case the continuation is observable, or by an unobservable behaviour otherwise.
2. *Coarse-grained bisimilarity* $\approx^{cg}$. Here, each converging sequence of steps must be simulated by a converging sequence of steps, at each instant. For instant changes, the requirement is the same as for fine-grained bisimulation.

As may be expected, the latter equivalence is more abstract than the former, as it only compares the I/O behaviours of programs (as functions on sets of events) at each instant, while the former also compares their intermediate results. Let us move now to the formal definitions of the equivalences $\approx^{fg}$ and $\approx^{cg}$.

Notation. • $\llcorner s \lrcorner_E \stackrel{\text{def}}{=} \begin{cases} [s]_E & \text{if } \langle s, E\rangle \ddagger \\ s & \text{otherwise} \end{cases}$ • $\langle s, E\rangle \overset{\ddagger}{\curlyvee} \Leftrightarrow \langle s, E\rangle \ddagger \vee s = \texttt{nothing}$

Definition 4 (Fine-grained bisimulation).
A symmetric relation $\mathcal{R}$ on programs is a fg-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any $E \subseteq Events$:*

$$1)\ \langle s_1, E\rangle \rightarrow \langle s_1', E'\rangle \;\;\Rightarrow\;\; \exists\, s_2' \,.\, (\, \langle s_2, E\rangle \Rightarrow \langle s_2', E'\rangle \;\wedge\; s_1' \,\mathcal{R}\, s_2' \,)$$

$$2)\ \langle s_1, E\rangle \ddagger \;\;\Rightarrow\;\; (\, \langle s_2, E\rangle \overset{\ddagger}{\curlyvee} \;\wedge\; \llcorner s_1 \lrcorner_E \;\mathcal{R}\; \llcorner s_2 \lrcorner_E \,)$$

Then s_1, s_2 are fg-bisimilar*, $s_1 \approx^{fg} s_2$, if $s_1 \,\mathcal{R}\, s_2$ for some fg-bisimulation $\mathcal{R}$.*

The equivalence $\approx^{fg}$ is *time-insensitive*, and thus insensitive to internal moves. It is also *termination-insensitive*, as it cannot distinguish proper termination from suspension (recall that no divergence is possible within an instant and thus the execution of a diverging program always spans over an infinity of instants). On the other hand, $\approx^{fg}$ is sensitive to the order of generation of events and to repeated emissions of the same event ("stuttering"). Typical examples are:

$$(\texttt{nothing}\,;\,\texttt{generate}\,ev) \;\approx^{fg}\; \texttt{generate}\,ev \;\not\approx^{fg}\; (\texttt{generate}\,ev\,;\,\texttt{generate}\,ev)$$

$$(\texttt{generate}\,ev_1 \nmid \texttt{generate}\,ev_2) \quad \not\approx^{fg} \quad (\texttt{generate}\,ev_2 \nmid \texttt{generate}\,ev_1)$$

$$\texttt{nothing} \quad \approx^{fg} \quad \texttt{cooperate} \quad \approx^{fg} \quad \texttt{loop}\,\texttt{nothing}$$

Definition 5 (Coarse-grained bisimulation).
A symmetric relation $\mathcal{R}$ on programs is a cg-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any $E \subseteq Events$:*

$$\langle s_1, E\rangle \Downarrow \langle s_1', E'\rangle \;\;\Rightarrow\;\; \exists\, s_2' \,.\, (\, \langle s_2, E\rangle \Downarrow \langle s_2', E'\rangle \;\wedge\; \llcorner s_1' \lrcorner_{E'} \;\mathcal{R}\; \llcorner s_2' \lrcorner_{E'} \,)$$

Then s_1, s_2 are cg-bisimilar*, $s_1 \approx^{cg} s_2$, if $s_1 \,\mathcal{R}\, s_2$ for some cg-bisimulation $\mathcal{R}$.*

Like $\approx^{fg}$, the equivalence $\approx^{cg}$ is both time-insensitive and termination-insensitive. Furthermore, it is also *stuttering-insensitive* and *generation-order-insensitive* (that is, it ignores the generation order of events within an instant). Typically:

$$\texttt{generate}\,ev \;\approx^{cg}\; (\texttt{generate}\,ev\,;\,\texttt{generate}\,ev)$$

$$(\texttt{generate}\,ev_1 \nmid \texttt{generate}\,ev_2) \quad \approx^{cg} \quad (\texttt{generate}\,ev_2 \nmid \texttt{generate}\,ev_1)$$

More generally, the equivalence $\approx^{cg}$ views $\nmid$ as a commutative operator:

Theorem 2. (Commutativity of $\nmid$ up to $\approx^{cg}$) $\forall s_1, s_2 \;\; (\, s_1 \nmid s_2 \approx^{cg} s_2 \nmid s_1 \,)$

Finally, we prove that $\approx^{fg}$ is strictly included in $\approx^{cg}$ (the strictness of the inclusion being witnessed by the last two examples above):

Theorem 3. (Relation between the bisimilarities) $\approx^{fg} \subset \approx^{cg}$

This concludes our discussion about semantic equivalences. We turn now to the definition of noninterference, which is grounded on that of bisimulation.

6 Security property

In this section we define two noninterference properties for programs. As usual when dealing with secure information flow, we assume a finite lattice $(\mathcal{S}, \leq)$ of *security levels*, ranged over by τ, σ, ϑ. We denote by $\sqcup$ and $\sqcap$ the join and meet operations on the lattice, and by $\perp$ and $\top$ its minimal and maximal elements.

In CRL, the objects that are assigned a security level are events and variables. An *observer* is identified with a downward-closed set of security levels (for short, a dc-set), i.e. a set $\mathcal{L} \subseteq \mathcal{S}$ satisfying the property: $(\tau \in \mathcal{L} \wedge \tau' \leq \tau) \Rightarrow \tau' \in \mathcal{L}$.

A type environment Γ is a mapping from variables and events to their types, which are just security levels τ, σ. Given a dc-set $\mathcal{L}$, a type environment Γ and an event environment E, the subset of E to which Γ assigns security levels in $\mathcal{L}$ is called the $\mathcal{L}$-*part* of E under Γ. Similarly, if $V : Var \to Val$ is a valuation, the subset of V whose domain is given levels in $\mathcal{L}$ by Γ is the $\mathcal{L}$-*part* of V under Γ.

Two event environments E_1, E_2 or two valuations V_1, V_2 are $=^{\Gamma}_{\mathcal{L}}$-equal, or indistinguishable by a $\mathcal{L}$-observer, if their $\mathcal{L}$-parts under Γ coincide:

Definition 6 ($\Gamma\mathcal{L}$-equality of event environments and valuations).

Let $\mathcal{L} \subseteq \mathcal{S}$ be a dc-set, Γ a type environment and V a valuation. Define:

$$E_1 =^{\Gamma}_{\mathcal{L}} E_2 \quad \textit{if} \quad \forall\, ev \in Events \;\; (\, \Gamma(ev) \in \mathcal{L} \;\Rightarrow\; (\, ev \in E_1 \Leftrightarrow ev \in E_2\,)\,)$$
$$V_1 =^{\Gamma}_{\mathcal{L}} V_2 \quad \textit{if} \quad \forall\, x \in Var \;\; (\, \Gamma(x) \in \mathcal{L} \;\Rightarrow\; V_1(x) = V_2(x)\,)$$

Let $\rightarrow_V, \Rightarrow_V, \Downarrow_V$ denote our various semantic arrows under the valuation V. Then we may define the indistinguishability of two programs by a fine-grained or coarse-grained $\mathcal{L}$-observer, for a given Γ, by means of the following two notions of $\Gamma\mathcal{L}$-bisimilarity:

Definition 7 (Fine-grained $\Gamma\mathcal{L}$-bisimilarity).

A symmetric relation $\mathcal{R}$ on programs is a fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$:*

1) $\langle s_1, E_1\rangle \rightarrow_{V_1} \langle s_1', E_1'\rangle \;\Rightarrow\; \exists\, s_2', E_2' \,.\, (\,\langle s_2, E_2\rangle \Rightarrow_{V_2} \langle s_2', E_2'\rangle \;\wedge\; E_1' =^{\Gamma}_{\mathcal{L}} E_2' \;\wedge\; s_1' \,\mathcal{R}\, s_2'\,)$

2) $\langle s_1, E_1\rangle \ddagger \;\;\Rightarrow\;\; (\,\langle s_2, E_2\rangle \overset{\ddagger}{\curlyvee} \;\wedge\; \llcorner s_1 \lrcorner_{E_1} \;\mathcal{R}\; \llcorner s_2 \lrcorner_{E_2}\,)$

Then programs s_1, s_2 are fg-$\Gamma\mathcal{L}$-bisimilar*, $s_1 \approx^{fg}_{\Gamma\mathcal{L}} s_2$, if for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, $s_1 \,\mathcal{R}\, s_2$ for some* fg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *$\mathcal{R}$.*

Definition 8 (Coarse-grained $\Gamma\mathcal{L}$-bisimilarity).

A symmetric relation $\mathcal{R}$ on programs is a cg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *if $s_1 \,\mathcal{R}\, s_2$ implies, for any E_1, E_2 such that $E_1 =^{\Gamma}_{\mathcal{L}} E_2$:*

$$\langle s_1, E_1\rangle \Downarrow_{V_1} \langle s_1', E_1'\rangle \;\Rightarrow\; \exists\, s_2', E_2' \,.\, (\;\langle s_2, E_2\rangle \Downarrow_{V_2} \langle s_2', E_2'\rangle \;\wedge\; E_1' =^{\Gamma}_{\mathcal{L}} E_2' \;\wedge\; \llcorner s_1' \lrcorner_{E_1'} \;\mathcal{R}\; \llcorner s_2' \lrcorner_{E_2'}\,)$$

Two programs s_1, s_2 are cg-$\Gamma\mathcal{L}$-bisimilar*, $s_1 \approx^{cg}_{\Gamma\mathcal{L}} s_2$, if for any V_1, V_2 such that $V_1 =^{\Gamma}_{\mathcal{L}} V_2$, $s_1 \,\mathcal{R}\, s_2$ for some* cg-$\Gamma\mathcal{L}$-V_1V_2-bisimulation *$\mathcal{R}$.*

Our *reactive noninterference* (RNI) properties are now defined as follows:

Definition 9 (Fine-grained and Coarse-grained RNI).

A program s is fg-secure in Γ if $s \approx^{fg}_{\Gamma\mathcal{L}} s$ for every dc-set $\mathcal{L}$.

A program s is cg-secure in Γ if $s \approx^{cg}_{\Gamma\mathcal{L}} s$ for every dc-set $\mathcal{L}$.

In examples, we use superscripts to indicate the level of variables and events.

Example 1. The following program is *cg*-secure but not *fg*-secure:

$$s = \texttt{if}\ x^{\top} = 0\ \texttt{then generate}\ ev_1^{\perp} \nmid \texttt{generate}\ ev_2^{\perp}$$
$$\texttt{else generate}\ ev_2^{\perp} \nmid \texttt{generate}\ ev_1^{\perp}$$

If we replace the second branch of s by $\texttt{generate}\ ev_1^{\perp}\ ;\ \texttt{generate}\ ev_2^{\perp}$, then we obtain a program s' that is both *fg*-secure and *cg*-secure.

In general, from all the equivalences/inequivalences in page 10 we may obtain secure/insecure programs for the corresponding RNI property by plugging the two equivalent/inequivalent programs in the branches of a high conditional.

Theorem 4. (Relation between the RNI properties)

Let $s \in Programs$. If s is fg-secure then s is cg-secure.

7 Type system

We present now our security type system for *CRL*, which is based on those introduced in [6] and [13] for a parallel while language and already adapted to a reactive language in [2]. The originality of these type systems is that they associate pairs (τ, σ) of security levels with programs, where τ is a lower bound on the level of "writes" and σ is an upper bound on the level of "reads". This allows the level of reads to be recorded, and then to be used to constrain the level of writes in the remainder of the program. In this way, it is possible to obtain a more precise treatment of *termination leaks*[3] than in standard type systems.

Recall that a type environment Γ is a mapping from variables and events to security levels τ, σ. Moreover, Γ associates a type of the form $\overrightarrow{\tau} \rightarrow \tau$ to functions, where $\overrightarrow{\tau}$ is a tuple of types $\tau_1, \ldots, \tau_n$. Type judgements for expressions and programs have the form $\Gamma \vdash exp : \tau$ and $\Gamma \vdash s : (\tau, \sigma)$ respectively.

The intuition for $\Gamma \vdash exp : \tau$ is that τ is an *upper bound* on the levels of variables occurring in exp. According to this intuition, subtyping for expressions is *covariant*. The intuition for $\Gamma \vdash s : (\tau, \sigma)$ is that τ is a *lower bound* on the levels of events generated in s (the "writes"of s), and σ is an *upper bound* on the levels of events awaited or watched in s and of variables tested in s (the "reads" or *guards* of s, formally defined in Definition 10). Accordingly, subtyping for programs is *contravariant* in its first component, and *covariant* in the second.

[3] Leaks due to different termination behaviours in the branches of a conditional. In classical parallel while languages, termination leaks may also arise in while loops. This is not possible in *CRL*, given the simple form of the `loop` construct.

The typing rules for expressions and programs are presented in Figure 7. The three rules that increase the guard type are (WATCHING), (REPEAT) and (COND1), and those that check it against the write type of the continuation are (SEQ), (REPEAT) and (LOOP). Note that there are two more rules for the conditional, factoring out the cases where either both branches are finite or both branches are infinite: indeed, in these cases no termination leaks can arise and thus it is not necessary to increase the guard level. In Rule (COND2), *FIN* denotes the set of finite or *terminating* programs, namely those built without using the constructs `await` ev and `loop`. In Rule (COND3), *INF* denotes the set of infinite or *nonterminating* programs, defined inductively as follows:

- $\texttt{loop}\, s \in INF$; – $s \in INF \Rightarrow \texttt{repeat}\, exp\, \texttt{do}\, s \in INF$;
- $s_1 \in INF \Rightarrow s_1; s_2 \in INF$; – $s_1 \in FIN \wedge s_2 \in INF \Rightarrow s_1; s_2 \in INF$;
- $s_1 \in INF \vee s_2 \in INF \Rightarrow s_1 \nmid s_2 \in INF$
- $s_1 \in INF \wedge s_2 \in INF \Rightarrow \texttt{if}\, exp\, \texttt{then}\, s_1\, \texttt{else}\, s_2 \in INF$

Note that $FIN \cup INF \subset Programs$. Examples of programs that are neither in FIN nor in INF are: `await` ev, `if` exp `then nothing else` (`loop` s), and `do` (`loop` s) `watching` ev.

We now prove that typability implies security via the classical steps:

Lemma 4 (Subject Reduction).

Let $\Gamma \vdash s : (\tau, \sigma)$. *Then* $\langle s, E \rangle \to \langle s', E' \rangle$ *implies* $\Gamma \vdash s' : (\tau, \sigma)$, *and* $\langle s, E \rangle \ddagger$ *implies* $\Gamma \vdash [s]_E : (\tau, \sigma)$.

Definition 10. (Guards and Generated Events)

1) For any s, $Guards(s)$ *is the union of the set of events* ev *such that* s *contains an* `await` ev *or a* `do` s' `watching` ev *instruction (for some* s'*), together with the set of variables* x *that occur in* s *as argument of a function or in the expression* exp *of an instruction* `if` exp `then` s_1 `else` s_2 *or* `repeat` exp `do` s *in* s.

2) For any s, $Gen(s)$ *is the set of events* ev *such that* `generate` ev *occurs in* s.

Lemma 5 (Guard Safety and Confinement).

1. *If* $\Gamma \vdash s : (\tau, \sigma)$ *then* $\Gamma(g) \le \sigma$ *for every* $g \in Guards(s)$;
2. *If* $\Gamma \vdash s : (\tau, \sigma)$ *then* $\tau \le \Gamma(ev)$ *for every* $ev \in Gen(s)$.

Theorem 5 (Typability $\Rightarrow$ Fine-grained RNI).

Let $s \in Programs$. *If* s *is typable in* Γ *then* s *is fg-secure in* Γ.

Note that programs s, s' of Example 1 are not typable (although *cg*-secure). We conclude with some examples illustrating the use of the rules for the conditional.

Example 2. The following programs s_i and s are all typable:

$s_1 =$ `if` $(x^\top = 0)$ `then await` $ev_1^\top$ `else cooperate`	type $(\top, \top)$
$s_2 =$ `if` $(x^\top = 0)$ `then nothing else cooperate`	type $(\top, \bot)$
$s_3 =$ `if` $(x^\top = 0)$ `then nothing else (loop nothing)`	type $(\top, \top)$
$s_4 =$ `if` $(x^\top = 0)$ `then (loop nothing) else (loop cooperate)`	type $(\top, \bot)$
$s =$ `generate` $ev_2^\bot$	type $(\bot, \bot)$

Then $s_2; s$ and $s_4; s$ are typable but not $s_1; s$ nor $s_3; s$.

$$(\textsc{Val})\quad \Gamma \vdash v : \bot \qquad (\textsc{Var})\quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad (\textsc{SubExp})\quad \frac{\Gamma \vdash exp : \sigma, \quad \sigma \le \sigma'}{\Gamma \vdash exp : \sigma'}$$

$$(\textsc{Fun})\quad \frac{\Gamma \vdash \overrightarrow{exp} : \overrightarrow{\tau}, \quad \Gamma(f) = \overrightarrow{\tau} \to \tau, \quad \forall i\,.\,\tau_i \le \tau}{\Gamma \vdash f(\overrightarrow{exp}) : \tau}$$

Typing rules for expressions

$$(\textsc{Nothing})\quad \Gamma \vdash \texttt{nothing} : (\top, \bot) \qquad (\textsc{Cooperate})\quad \Gamma \vdash \texttt{cooperate} : (\top, \bot)$$

$$(\textsc{Seq})\quad \frac{\Gamma \vdash s_1 : (\tau_1, \sigma_1), \quad \Gamma \vdash s_2 : (\tau_2, \sigma_2), \quad \sigma_1 \le \tau_2}{\Gamma \vdash s_1\,;\,s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)}$$

$$(\textsc{Par})\quad \frac{\Gamma \vdash s_1 : (\tau_1, \sigma_1), \quad \Gamma \vdash s_2 : (\tau_2, \sigma_2)}{\Gamma \vdash s_1 \nmid s_2 : (\tau_1 \sqcap \tau_2, \sigma_1 \sqcup \sigma_2)}$$

$$(\textsc{Generate})\quad \frac{\Gamma(ev) = \tau}{\Gamma \vdash \texttt{generate}\,ev : (\tau, \bot)} \qquad (\textsc{Await})\quad \frac{\Gamma(ev) = \sigma}{\Gamma \vdash \texttt{await}\,ev : (\top, \sigma)}$$

$$(\textsc{Watching})\quad \frac{\Gamma(ev) = \vartheta, \quad \Gamma \vdash s : (\tau, \sigma), \quad \vartheta \le \tau}{\Gamma \vdash \texttt{do}\,s\,\texttt{watching}\,ev : (\tau, \vartheta \sqcup \sigma)}$$

$$(\textsc{Loop})\quad \frac{\Gamma \vdash s : (\tau, \sigma), \quad \sigma \le \tau}{\Gamma \vdash \texttt{loop}\,s : (\tau, \sigma)} \qquad (\textsc{Repeat})\quad \frac{\Gamma \vdash exp : \vartheta \quad \Gamma \vdash s : (\tau, \sigma), \quad \vartheta \sqcup \sigma \le \tau}{\Gamma \vdash \texttt{repeat}\,exp\,\texttt{do}\,s : (\tau, \vartheta \sqcup \sigma)}$$

$$(\textsc{Cond1})\quad \frac{\Gamma \vdash exp : \vartheta, \quad \Gamma \vdash s_i : (\tau, \sigma), \quad i = 1, 2, \quad \vartheta \le \tau}{\Gamma \vdash \texttt{if}\,exp\,\texttt{then}\,s_1\,\texttt{else}\,s_2 : (\tau, \vartheta \sqcup \sigma)}$$

$$(\textsc{Cond2})\quad \frac{\Gamma \vdash exp : \vartheta, \quad (\Gamma \vdash s_i : (\tau, \sigma) \quad \wedge \quad s_i \in FIN, \quad i = 1, 2), \quad \vartheta \le \tau}{\Gamma \vdash \texttt{if}\,exp\,\texttt{then}\,s_1\,\texttt{else}\,s_2 : (\tau, \sigma)}$$

$$(\textsc{Cond3})\quad \frac{\Gamma \vdash exp : \vartheta, \quad (\Gamma \vdash s_i : (\tau, \sigma) \quad \wedge \quad s_i \in INF, \quad i = 1, 2), \quad \vartheta \le \tau}{\Gamma \vdash \texttt{if}\,exp\,\texttt{then}\,s_1\,\texttt{else}\,s_2 : (\tau, \sigma)}$$

$$(\textsc{SubProg})\quad \frac{\Gamma \vdash s : (\tau, \sigma), \quad \tau' \le \tau, \quad \sigma \le \sigma'}{\Gamma \vdash s : (\tau', \sigma')}$$

Typing rules for programs

Fig. 2. Security type system

8 Conclusion

We have studied a core reactive language CRL and proposed two RNI properties for it, together with a security type system ensuring them. We have established a reactivity result similar to those of [8,3], but based on different design choices. Our RNI properties rely on two bisimulation equivalences of different granularity. One of them, coarse-grained bisimilarity, is reminiscent of the semantic equivalence studied by Amadio in [3], which however was based on trace semantics. Our RNI properties bear some analogy with the *reactive noninterference* notions proposed in [5], although the underlying assumptions of the model are quite different (neither reactivity nor determinacy are assumed in [5], and there is no internal parallelism). The model of cooperative threads of [1] is close in spirit to the model of CRL, but it is not concerned with synchronous parallelism.

We plan to extend our results to a fully-fledged distributed reactive language.

Acknowledgments

We thank Frédéric Boussinot for insightful discussions and helpful feedback.

References

1. M. Abadi and G. Plotkin. A model of cooperative threads. In *ACM SIGPLAN Notices*, volume 44, pages 29–40. ACM, 2009.
2. A. Almeida Matos, G. Boudol, and I. Castellani. Typing noninterference for reactive programs. *Journal of Logic and Algebraic Programming*, 72(2):124–156, 2007.
3. Roberto M Amadio. The SL synchronous language, revisited. *The Journal of Logic and Algebraic Programming*, 70(2):121–150, 2007.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comput. Programming*, 19, 1992.
5. A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 79–90. ACM, 2009.
6. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.
7. F. Boussinot and R. de Simone. The SL synchronous language. *Software Engineering*, 22(4):256–266, 1996.
8. F. Boussinot and J-F. Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software: Practice and Experience*, 28(14):1531–1550, 1998.
9. R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design - Tutorial Lectures*, number 2171 in LNCS, 2001.
10. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
11. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, 1984.
12. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
13. G. Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. IEEE, 2001.
14. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.