The SL synchronous language, revisited

Roberto M. Amadio

To cite this version:

Roberto M. Amadio. The SL synchronous language, revisited. Journal of Logic and Algebraic Programming, Elsevier, 2007, 70, pp.121-150. <hal-00014540>

HAL Id: hal-00014540

https://hal.archives-ouvertes.fr/hal-00014540

Submitted on 28 Nov 2005

The SL synchronous language, revisited

Roberto M. Amadio*
Université Paris 7†

28th November 2005

Abstract

We revisit the SL synchronous programming model introduced by Boussinot and De Simone *(IEEE, Trans. on Soft. Eng., 1996)*. We discuss an alternative design of the model including *thread spawning* and *recursive definitions* and we explore some basic properties of the revised model: determinism, reactivity, CPS translation to a tail recursive form, computational expressivity, and a compositional notion of program equivalence.

1 Introduction

In synchronous models the computation of a set of participants is regulated by a notion of *instant*. The *Synchronous Language* introduced in [8] belongs to this category. A *program* in this language generally contains sub-programs running in parallel and interacting via shared *signals*. By default, at the beginning of each instant a signal is absent and once it is emitted it remains in that state till the end of the instant. The model can be regarded as a relaxation of the ESTEREL model [5] where the *reaction to the absence* of a signal is delayed to the following instant, thus avoiding the difficult problems due to *causality cycles* in ESTEREL programs.

The model has gradually evolved into a programming language for concurrent applications and has been implemented in the context of various programming languages such as C, JAVA, SCHEME, and CAML (see, *e.g.*, [19, 20, 13]). The design accommodates a dynamic computing environment with threads entering or leaving the synchronisation space [6]. In this context, it seems natural to suppose that the scheduling of the threads is only determined at run time (as opposed to certain synchronous languages such as ESTEREL or LUSTRE). It appears that many typical "concurrent" applications such as event-driven controllers, data flow architectures, graphical user interfaces, simulations, web services, multiplayer games, are more effectively programmed in the synchronous framework.

*Partially supported by ACI *Sécurité Informatique* CRISS.
†Laboratoire *Preuves, Programmes et Systèmes*, UMR-CNRS 7126.

The SL language was carefully designed to be compiled to finite state automata. Motivated by the evolution of the language mentioned above, we consider a *synchronous language* including *thread spawning*, and *recursive definitions* (section 2) and we explore some basic properties of the revised model. First, we prove that the resulting language is deterministic and provide a simple static analysis that entails reactivity (section 3). Second, we propose a continuation passing style translation to a more basic language of tail recursive threads (section 4). Third, we show that the language without signal generation has the same computational power as a class of 'monotonic' Mealy machines, while the language with signal generation is Turing equivalent (section 5). Fourth, we introduce a notion of contextual barbed bisimulation and characterise it via a suitable labelled bisimulation (section 6). Some standard proofs are delayed to the appendix A.

1.1 Related work

This work is a continuation of [1] where we outline results and problems connected with the SL model 10 years after its proposal. A determinacy theorem was already stated in the original paper [8] with a similar proof based on the confluence of the 'small step' reduction. Of course, many other determinacy theorems occur in the literature on synchronous programming (cf., *e.g.*, [12]). The static analysis technique for ensuring reactivity is inspired by previous work by the author [3, 4] where, roughly, the reactivity of a (tail recursive) SL model with data types is studied. The tail recursive SL model and the related CPS translation appear to be original. They arose out of an attempt to understand the relative expressivity of various synchronous operators such as await, when and watch. The results on the computational expressivity of the revised model, notably its characterisation via monotonic Mealy machine, were motivated by the compilation to finite state machines in the original SL proposal [8]. Finally, there seems to be no previous attempt at developing a compositional notion of bisimulation equivalence for the SL model in a CCS style. However a specific notion of bisimulation for 'closed systems' has been proposed recently in the framework of the work on non-interference for synchronous systems [14].

2 The model

In this section, we present a formalisation of the model which is largely inspired by the original proposition [8] and a recent survey [1]. We anticipate that in section 4 we will simplify the control structure by moving to a tail recursive model and in section 6 we will discuss an alternative presentation in the spirit of process calculi.

2.1 Environments

We assume a countable set S of *signal names* $s, s', \ldots$. We suppose a subset $\mathit{Int} = \mathit{Input} \cup \mathit{Output}$ of S of *observable* signal names representing input or output signals and such that $S \backslash \mathit{Int}$ is infinite. An *environment* E is a partial function from signal names to boolean values *true* and *false* whose domain of definition $dom(E)$ contains Int and such that $S \backslash dom(E)$ is infinite.

2.2 Threads

We denote with $\mathbf{x}$ a vector of elements $x_1, \ldots, x_n$, $n \geq 0$ and with $[\_/\_]$ the usual substitution. By default, bound names can be renamed. We denote with $A(\mathbf{s}), B(\mathbf{s}), \ldots$ thread identifiers with parameters $\mathbf{s}$. As usual, each thread identifier is defined by exactly one equation $A(\mathbf{x}) = T$ where T is a *thread* defined by the grammar:

$$T ::= 0 \mid (T;T) \mid (\mathsf{emit}\ s) \mid (\nu s\ T) \mid (\mathsf{thread}\ T) \mid (\mathsf{await}\ s) \mid (\mathsf{watch}\ s\ T) \mid A(\mathbf{s})$$

and the signal names free in T are contained in $\{\mathbf{x}\}$. Sometimes, some of the parameters (possibly all) are fixed and in these cases we will feel free to omit them. A thread is executed relatively to an environment which is *shared* with other parallel threads. The intended semantics is as follows: 0 is the terminated thread; $T;T$ is the usual sequentialisation; $(\mathsf{emit}\ s)$ emits s, *i.e.* sets to *true* the signal s and terminates, $(\nu s\ T)$ creates a fresh signal which is local to the thread T (s is bound in T) and executes T; $(\mathsf{thread}\ T)$ spawns a thread T which will be executed in parallel and terminates; $(\mathsf{await}\ s)$ terminates if the signal s is present and suspends the execution otherwise; $(\mathsf{watch}\ s\ T)$ allows the execution of T but terminates T at the end of the first instant where the signal s is present. The implementation of the watch instruction requires to stack the signals that may cause the abortion of the current thread together with the associated continuations. For instance, in $(\mathsf{watch}\ s_1\ (\mathsf{watch}\ s_2\ T_1);T_2);T_3$, we start executing T_1. Assuming that at the end of the instant, the execution of T_1 is not completed, the computation in the following instant resumes with T_3 if s_1 was present at the end of the instant, with T_2 if s_1 was absent and s_2 was present at the end of the instant, and with the residual of T_1, otherwise. We point out that a thread spawned by the thread instruction, escapes the watch signals and the related continuations.

2.3 Thread reduction

A *program* P is a finite non-empty *multi-set* of threads. We denote with $sig(T)$ ($sig(P)$) the set of signals free in T (in threads in P). Whenever we write (T, E), (P, E) it is intended that $sig(T) \subseteq dom(E)$, $sig(P) \subseteq dom(E)$, respectively. All reduction rules maintain the invariant that the signals defined in the thread or in the program are in the domain of definition of the associated environment. In particular, all signal names which are not in the domain of definition of the environment are guaranteed to be *fresh*, i.e., not used elsewhere in the program. Finally, we make the usual assumption that reduction rules are given modulo renaming of the bound signal names.

We assume that sequential composition ';' associates to the right. A *redex* Δ is defined by the grammar:

$$\Delta ::= 0;T \mid (\mathsf{emit}\ s) \mid (\nu s\ T) \mid (\mathsf{thread}\ T) \mid (\mathsf{await}\ s) \mid (\mathsf{watch}\ s\ 0) \mid A(\mathbf{s})\ .$$

An *evaluation context* C is defined by the grammar:

$$C ::= [\,] \mid [\,];T \mid (\mathsf{watch}\ s\ C) \mid (\mathsf{watch}\ s\ C);T\ .$$

We have a canonical decomposition of a thread in an evaluation context and a redex whose proof is delayed to appendix A.1.

Proposition 1 (unique decomposition) *A thread $T \neq 0$ admits a unique decomposition $T = C[\Delta]$ into an evaluation context C and a redex Δ. Moreover, if $T = 0$ then no decomposition exists.*

The reduction relation $(T, E) \xrightarrow{P} (T', E')$ is defined first on redexes by the rules (T_{1-7}) and then it is lifted to threads by the rule (T_8):

$$\begin{array}{llll}
(T_1) & (0;T, E) & \xrightarrow{\emptyset} (T, E) & \\
(T_2) & (\mathsf{emit}\ s, E) & \xrightarrow{\emptyset} (0, E[\mathit{true}/s]) & \\
(T_3) & (\mathsf{watch}\ s\ 0, E) & \xrightarrow{\emptyset} (0, E) & \\
(T_4) & (\nu s\ T, E) & \xrightarrow{\emptyset} (T, E[\mathit{false}/s]) & \text{if } s \notin \mathit{dom}(E) \\
(T_5) & (A(\mathbf{s}), E) & \xrightarrow{\emptyset} ([\mathbf{s}/\mathbf{x}]T, E) & \text{if } A(\mathbf{x}) = T \\
(T_6) & (\mathsf{await}\ s, E) & \xrightarrow{\emptyset} (0, E) & \text{if } E(s) = \mathit{true} \\
(T_7) & (\mathsf{thread}\ T, E) & \xrightarrow{\{\!|T|\!\}} (0, E) & \\
(T_8) & (C[\Delta], E) & \xrightarrow{P} (C[T'], E') & \text{if } (\Delta, E) \xrightarrow{P} (T', E')
\end{array}$$

We write $(T, E) \downarrow$ if T cannot be reduced in the environment E according to the rules above. We also say that (T, E) is *suspended*. An inspection of the rules reveals that $(T, E) \downarrow$ if and only if $T = 0$ or $T = C[(\mathsf{await}\ s)]$ with $E(s) = \mathit{false}$. Thus the await statement is the only one that may cause the suspension of a thread. The suspension predicate is extended to programs as follows $(P, E) \downarrow$ if $\forall T \in P\ \ (T, E) \downarrow$.

2.4 Program reduction

To execute a program P in an environment E during an instant proceed as follows:
(1) Schedule (non-deterministically) the executions of the threads that compose it as long as some progress is possible according to the rule:

$$(P \cup \{\!|T|\!\}, E) \rightarrow (P \cup \{\!|T'|\!\} \cup P'', E') \quad \text{if} \quad (T, E) \xrightarrow{P''} (T', E')\ .$$

We also write $(P \cup \{\!|T|\!\}, E) \xrightarrow{P''} (P \cup \{\!|T'|\!\}, E')$ if $(T, E) \xrightarrow{P''} (T', E')$.
(2) Transform all (watch s T) instructions where the signal s is present into the terminated thread 0. Formally, we rely on the function $\lfloor \_ \rfloor_E$ defined on a multiset of suspended threads as follows:

$$\lfloor P \rfloor_E = \{\!| \lfloor T \rfloor_E \mid T \in P |\!\} \quad \lfloor 0 \rfloor_E = 0 \quad \lfloor T;T' \rfloor_E = \lfloor T \rfloor_E;T' \quad \lfloor \mathsf{await}\ s \rfloor_E = (\mathsf{await}\ s)$$

$$\lfloor \mathsf{watch}\ s\ T \rfloor_E = \begin{cases} 0 & \text{if } E(s) = \mathit{true} \\ (\mathsf{watch}\ s\ \lfloor T \rfloor_E) & \text{otherwise} \end{cases}$$

2.5 Trace semantics

Finally, the input-output behaviour of a program is described by labelled transitions $P \overset{I/O}{\to} P'$ where $I \subseteq \mathit{Input}$ and $O \subseteq \mathit{Output}$ are the signals in the interface which are present in input at the beginning of the instant and in output at the end of the instant, respectively. As in Mealy machines, the transition means that from program (state) P with 'input' signals I we move to program (state) P' with 'output' signals O. This is formalised by the rule:

$$(I/O) \quad \frac{(P, E_{I,P}) \overset{*}{\to} (P', E'), \quad (P', E') \downarrow, \quad O = \{s \in \mathit{Output} \mid E'(s) = \mathit{true}\}}{P \overset{I/O}{\to} P'}$$

$$\text{where:} \quad E_{I,P}(s) = \begin{cases} \mathit{true} & \text{if } s \in I \\ \mathit{false} & \text{if } s \in (\mathit{Int} \cup \mathit{sig}(P)) \backslash I \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

Note that in the definition of $E_{I,P}$ we insist on having all signals free in the program in the domain of definition of the environment and we leave the others undefined so that they can be potentially used in the rule (T_4). A *complete* run of a program P is a reduction $P \overset{I_1/O_1}{\to} P_1 \overset{I_2/O_2}{\to} P_2 \cdots$ which is either infinite or is finite and cannot be further extended. We define an extensional semantics of a program P, as the set $\mathit{tr}(P)$ of (finite or infinite) words associated with its complete runs. Namely:

$$\mathit{tr}(P) = \{(I_1/O_1)(I_2/O_2)\cdots \mid I_j \subseteq \mathit{Input}, O_j \subseteq \mathit{Output}, P \overset{I_1/O_1}{\to} P_1 \overset{I_2/O_2}{\to} P_2 \cdots\} \qquad (1)$$

2.6 Derived instructions

We may abbreviate $(\nu s_1 \cdots (\nu s_n\, T) \cdots)$ as $(\nu s_1, \ldots, s_n\, T)$ and $(\mathsf{thread}\ T_1); \cdots (\mathsf{thread}\ T_n)$ as $(\mathsf{thread}\ T_1, \ldots, T_n)$. Table 1 presents some derived instructions which are frequently used in the programming practice. The instruction $(\mathsf{loop}\ T)$ can be thought as $T; T; T; \cdots$. Note that in $(\mathsf{loop}\ T); T'$, T' is *dead code*, *i.e.*, it can never be executed. The instruction $(\mathsf{now}\ T)$ runs T for the current instant, *i.e.*, if the execution of T is not completed within the current instant then it is aborted. The instruction pause suspends the execution of the thread for the current instant and resumes it in the following one. We will rely on this instruction to guarantee the termination of the computation of each thread within an instant (see section 3). The instruction $(\mathsf{present}\ s\ T_1\ T_2)$ branches on the presence of a signal. Note that the branch T_2 corresponding to the *absence* of the signal is executed in the following instant and that we suppose $s' \notin \mathit{sig}(T_1) \cup \mathit{sig}(T_2)$. The instruction $(T_1 \parallel T_2)$ runs in parallel the threads T_1 and T_2 and waits for their termination. Here we suppose that $s_1, s_2, s'_1, s'_2 \notin \mathit{sig}(T_1) \cup \mathit{sig}(T_2)$.

2.7 Comparison with [8]

The main novelty with respect to [8] is the replacement of loop and parallel composition operators with recursive definitions and thread spawning. We should stress that the en-

$$
\begin{array}{ll}
(\mathsf{loop}\ T) & = A \quad \text{where: } A\ =\ T; A \\
(\mathsf{now}\ T) & = \nu s\ (\mathsf{emit}\ s); (\mathsf{watch}\ s\ T) \quad s \notin sig(T) \\
\mathsf{pause} & = \nu s\ (\mathsf{now}\ (\mathsf{await}\ s)) \\
(\mathsf{present}\ s\ T_1\ T_2) & = \nu s'\ (\mathsf{thread} \\
& \qquad\qquad (\mathsf{now}\ (\mathsf{await}\ s); (\mathsf{thread}\ T_1; (\mathsf{emit}\ s'))), \\
& \qquad\qquad (\mathsf{watch}\ s\ \mathsf{pause}; (\mathsf{thread}\ T_2; (\mathsf{emit}\ s')))\ ); (\mathsf{await}\ s') \\
(T_1 \parallel T_2) & = \nu s_1, s_2, s_1', s_2'\ (\mathsf{thread} \\
& \qquad\qquad (\mathsf{watch}\ s_1'\ T_1; (\mathsf{loop}\ (\mathsf{emit}\ s_1); \mathsf{pause})), \\
& \qquad\qquad (\mathsf{watch}\ s_2'\ T_2; (\mathsf{loop}\ (\mathsf{emit}\ s_2); \mathsf{pause}))\ ); \\
& \qquad\qquad\quad (\mathsf{await}\ s_1); (\mathsf{emit}\ s_1'); (\mathsf{await}\ s_2); (\mathsf{emit}\ s_2')
\end{array}
$$

Table 1: Some derived instructions

coding of the present and parallel composition operators do not correspond exactly to the operators in the original language. This is because the instructions T_1 and T_2 are under a thread instruction and therefore their execution does *not* depend on watch signals that may be on top of them. If this must be the case, then we must prefix T_1 and T_2 with suitable watch instructions. The CPS translation discussed in section 4, provides a systematic method to simulate the stack of watch signals.

2.8 Cooperative vs. preemptive concurrency

In *cooperative* concurrency a running thread cannot be interrupted unless it explicitly decides to return the control to the scheduler. This is to be contrasted with *preemptive* concurrency where a running thread can be interrupted at any point unless it explicitly requires that a series of actions is atomic (*e.g.*, via a lock). We refer to, *e.g.*, [17] for an extended comparison of the cooperative and preemptive models in the practice of programming. In its original proposal, the SL language adopts a cooperative notion of concurrency. Technically this means that a 'big step' reduction is defined on top of the 'small step' reduction we have introduced. The big step reduction runs a thread atomically till it terminates or it suspends on an await statement. Programs are then evaluated according to this big step reduction. In particular, this means that the small step reductions cannot be freely interleaved. In the following, we will focus on the small step/preemptive semantics and neglect the big step/cooperative semantics for two reasons: (1) All main results (determinism, reactivity, CPS translation) are naturally obtained at the level of the small step/preemptive semantics and are then lifted to the big step/cooperative semantics. (2) The cooperative semantics goes against the natural idea of executing a program with parallel threads on a multi-processor where the threads run in parallel on different processors up to a synchronisation point.

3 Determinism and reactivity

We consider two important properties a SL program should have: *determinism* and *reactivity*. While the first property is ensured by the design of the language (as was the case in the original language), we enforce the second by means of a new static analysis.

3.1 Determinism

It is immediate to verify that the evaluation of a thread T in an environment E is deterministic. Therefore the only potential source of non-determinism comes from the scheduling of the threads. The basic remark is that the emission of a signal can never block the execution of a statement within an instant. The more signals are emitted the more the computation of a thread can progress within an instant. Of course, this *monotonicity property* relies on the fact that a thread cannot detect the absence of a signal before the end of an instant.

Technically, the property that entails determinism is the fact that the small step reduction is strongly confluent up to *renaming*. A renaming σ is a bijection σ on signal names which is the identity on the names in the interface *Int*. We introduce a notion of *equality up to renaming*: (i) $T =_\alpha T'$ if there is a renaming σ such that $\sigma T = T'$ and (ii) $(T, E) =_\alpha (T', E')$ if there is a renaming σ such that $\sigma T = T'$ and $E = E' \circ \sigma$. In a similar way, we define $P =_\alpha P'$ and $(P, E) =_\alpha (P', E')$. We rely on equality up to renaming to define a notion of determinism.

Definition 2 *The set of* deterministic *programs is the largest set of programs $\mathcal{D}$ such that if $P \in \mathcal{D}$, $I \subseteq$ Input, $P \overset{I/O_1}{\rightarrow} P_1$, and $P \overset{I/O_2}{\rightarrow} P_2$ then $O_1 = O_2$ and $P_1 =_\alpha P_2 \in \mathcal{D}$.*

In appendix A.2, we show how to derive determinism from strong confluence by means of a standard tiling argument.

Theorem 3 *All programs are deterministic.*

3.2 Reactivity

We now turn to a formal definition of reactivity.

Definition 4 *The set of* reactive *programs is the largest set of programs $\mathcal{R}$ such that if $P \in \mathcal{R}$ then for every choice $I \subseteq$ Input of the input signals there are O, P' such that $P \overset{I/O}{\rightarrow} P'$ and $P' \in \mathcal{R}$.*

We can write programs which are not reactive. For instance, the thread $A = (\mathsf{await}\ s); A$ may potentially loop within an instant. Whenever a thread loops within an instant the computation of the whole program is blocked as the instant never terminates. In the programming practice, reactivity is ensured by instrumenting the code with pause statements that force the computation to suspend for the current instant. Following this practice, we take the pause statement as a primitive, though it can can be defined as seen in section 2.6. This can be easily done by observing that a suspended thread may also have the shape $C[\mathsf{pause}]$ and by extending the evaluation at the end of the instant with the equation $\lfloor \mathsf{pause} \rfloor_E = 0$. We introduce next a *static analysis* that guarantees reactivity on a code with explicit pause statements.

We denote with $X, Y, \ldots$ finite multisets of thread identifiers and with ℓ a label ranging over the symbols 0 and $\downarrow$. We define a function *Call* associating with a thread T a pair (X, ℓ) where intuitively the multi-set X represents the thread identifiers that T may call within the current instant and ℓ indicates whether a continuation of T has the possibility of running within the current instant ($\ell = 0$) or not ($\ell =\downarrow$). As usual, π_i projects a tuple on the i^{th} component.

$$\begin{array}{cc}
\mathit{Call}(0) = \mathit{Call}(\mathsf{emit}\ s) = \mathit{Call}(\mathsf{await}\ s) = (\emptyset, 0) & \mathit{Call}(\mathsf{pause}) = (\emptyset, \downarrow) \\
\mathit{Call}(\nu s\ T) = \mathit{Call}(\mathsf{watch}\ s\ T) = \mathit{Call}(T) & \mathit{Call}(A(\mathbf{s})) = (\{\!|A|\!\}, 0) \\
\mathit{Call}(\mathsf{thread}\ T) = (\pi_1(\mathit{Call}(T)), 0) & \mathit{Call}(T_1; T_2) = \mathit{Call}(T_1); \mathit{Call}(T_2)
\end{array}$$

where the operation ';' is defined on the codomain of *Call* as follows:

;	$(Y, 0)$	$(Y, \downarrow)$
$(X, 0)$	$(X \cup Y, 0)$	$(X \cup Y, \downarrow)$
$(X, \downarrow)$	$(X, \downarrow)$	$(X, \downarrow)$

We notice that this operation is *associative*. It is convenient to define the *Call* function also on evaluation contexts as follows:

$$\begin{array}{cc}
\mathit{Call}([\ ]) = \emptyset & \mathit{Call}([\ ]; T) = \mathit{Call}(T) \\
\mathit{Call}(\mathsf{watch}\ s\ C) = \mathit{Call}(C) & \mathit{Call}((\mathsf{watch}\ s\ C); T') = \mathit{Call}(C); \mathit{Call}(T')
\end{array}$$

and observe the following property which is proved by induction on the structure of the context.

Proposition 5 *For every evaluation context C and thread T, $\mathit{Call}(C[T]) = \mathit{Call}(T); \mathit{Call}(C)$.*

We can now introduce a static condition that guarantees reactivity. Intuitively, to ensure the reactivity of a program P, it is enough to find an *acyclic precedence relation* on the related thread identifiers which is consistent with their definitions. Namely, we define:

$$\mathit{Cnst}(P) = \{A > B \mid A(\mathbf{x}) = T \text{ equation for program } P, B \in \pi_1(\mathit{Call}(T))\}$$

Theorem 6 *A program P is reactive if there is a well founded order $>$ on thread identifiers that satisfies the inequalities in $\mathit{Cnst}(P)$.*

PROOF. The order $>$ on thread identifiers induces a well founded order on the finite multisets of thread identifiers. We denote this order with $>_{m,Id}$. We define a *size function sz* from threads to natural number $\mathbf{N}$ as follows:

$$\begin{array}{c}
sz(0) = sz(\mathsf{pause}) = 0, \quad sz(\mathsf{emit}\ s) = sz(\mathsf{await}\ s) = sz(A(\mathbf{s})) = 1, \\
sz(\nu s\ T) = sz(\mathsf{watch}\ s\ T) = sz(\mathsf{thread}\ T) = 1 + sz(T), \quad sz(T_1; T_2) = 1 + sz(T_1) + sz(T_2)
\end{array}$$

We denote with $>_{lex}$ the lexicographic order from left to right induced by the order $>_{m,Id}$ and the standard order on natural numbers. This order is well-founded. Finally, we consider the multi-set order $>_m$ induced by $>_{lex}$ on finite multi-sets. Again, this order is well founded. Next, we define a 'measure' μ associating with a program a finite multi-set:

$$\mu(P) = \{\!|(\pi_1(\mathit{Call}(T)), sz(T)) \mid T \in P|\!\} \ .$$

It just remains to check that the small step reduction decreases this measure. Namely, if $(P, E) \xrightarrow{P''} (P', E')$ then $\mu(P) >_m \mu(P') \cup \mu(P'')$, where the $\cup$ is of course intended on multi-sets. We recall that in the multi-set order an element can be replaced by a finite multi-set of strictly smaller elements. We proceed by case analysis on the small step reduction.

• Suppose the program reduction is induced by the thread reduction:

$$(C[\Delta], E) \xrightarrow{\emptyset} (C[T], E) \ .$$

where Δ has the shape $0; T'$, $\mathsf{emit}\ s$, $\nu s\ T'$, $\mathsf{await}\ s$, or $\mathsf{watch}\ s\ 0$. In these cases the first component does not increase while the size decreases.

• Suppose the program reduction is induced by the thread reduction:

$$(C[(\mathsf{thread}\ T)], E) \xrightarrow{\{\!|T|\!\}} (C[0], E) \ .$$

Assume $\mathit{Call}(T) = (X, \ell)$ and $\mathit{Call}(C) = (Y, \ell')$. By proposition 5, we have:

$$\begin{array}{c} \mathit{Call}(C[\mathsf{thread}\ T]) = \mathit{Call}(\mathsf{thread}\ T); \mathit{Call}(C) = (X, 0); (Y, \ell') = (X \cup Y, \ell') \\ \mathit{Call}(C[0]) = \mathit{Call}(0); \mathit{Call}(C) = (Y, \ell') \ . \end{array}$$

Thus the first component does not increase while the size decreases.

• Finally, suppose the program reduction comes from the unfolding of a recursive definition $A(\mathbf{x}) = T$:

$$C[A(\mathbf{s})] \xrightarrow{\emptyset} C[[\mathbf{s}/\mathbf{x}]T] \ .$$

Assume $\mathit{Call}(T) = (X, \ell)$ and $\mathit{Call}(C) = (Y, \ell')$. Then

$$\mathit{Call}(C[A(\mathbf{s})]) = (\{\!|A|\!\} \cup Y, \ell), \quad \mathit{Call}(C[T]) = \mathit{Call}(T); \mathit{Call}(C) = (X, \ell); (Y, \ell') \ .$$

By hypothesis, $\{\!|A|\!\} > X$. We derive that $\{\!|A|\!\} \cup Y >_{m,Id} X \cup Y \geq_{m,Id} Y$, and we notice that $(X, \ell); (Y, \ell')$ equals $(X \cup Y, \ell')$ if $\ell = 0$ and $(X, \downarrow)$, otherwise. □

Theorem 6 provides a sufficient (but not necessary) criteria to ensure reactivity.

Example 7 *Theorem 6 provides a sufficient (but not necessary) criteria to ensure reactivity. Indeed, the precision of the analysis can be improved by unfolding some recursive equations. For instance, consider the thread A defined by the system:*

$$\begin{array}{ll} A & = (\mathsf{watch}\ s_1\ B); (\mathsf{emit}\ s_4); A \\ B & = (\mathsf{await}\ s_2); (\mathsf{emit}\ s_3); \mathsf{pause}; B \end{array}$$

If we compute the corresponding Call we obtain:

$$\begin{array}{lll} \mathit{Call}((\mathsf{watch}\ s_1\ B);(\mathsf{emit}\ s_4);A) & =(\{\!|B|\!\},0);(\emptyset,0);(\{\!|A|\!\},0) & =(\{\!|A,B|\!\},0) \\ \mathit{Call}((\mathsf{await}\ s_2);(\mathsf{emit}\ s_3);\mathsf{pause};B) & =(\emptyset,0);(\emptyset,0);(\emptyset,\downarrow);(\{\!|B|\!\},0) & =(\emptyset,\downarrow) \end{array}$$

and obviously we cannot find a well founded order such that $A > A$. However, if we unfold B definition in A then we obtain $(\emptyset,\downarrow);(\emptyset,0);(\{\!|A|\!\},0) = (\emptyset,\downarrow)$, and the constraints are trivially satisfied.

4 A tail-recursive model and a CPS translation

We introduce a more basic language of *tail recursive threads* to which the 'high level language' introduced in section 2 can be compiled via a continuation passing style (CPS) translation. Tail recursive threads are denoted by $t, t', \ldots$ and they are defined as follows

$$t ::= 0 \mid A(\mathbf{s}) \mid \mathsf{emit}\ s.t \mid \nu s\ t \mid \mathsf{thread}\ t.t \mid \mathsf{present}\ s\ t\ b$$

where A is a thread identifier with the usual conventions (cf. section 2). Let $b, b', \ldots$ stand for *branching threads* defined as follows.

$$b ::= t \mid \mathsf{ite}\ s\ b\ b$$

Branching threads can only occur in the 'else' branch of a present instruction and they are executed only at the end of an instant once the presence or absence of a signal has been established. The small step thread reduction can be simply defined as follows:

$$\begin{array}{llll} (t_1) & (\mathsf{emit}\ s.t, E) & \xrightarrow{\emptyset} (t, E[\mathit{true}/s]) & \\ (t_2) & (\nu s\ t, E) & \xrightarrow{\emptyset} (t, E[\mathit{false}/s]) & \text{if } s \notin \mathit{dom}(E) \\ (t_3) & (A(\mathbf{s}), E) & \xrightarrow{\emptyset} ([\mathbf{s}/\mathbf{x}]t, E) & \text{if } A(\mathbf{x}) = t \\ (t_4) & (\mathsf{present}\ s\ t\ b, E) & \xrightarrow{\emptyset} (t, E) & \text{if } E(s) = \mathit{true} \\ (t_5) & (\mathsf{thread}\ t'.t, E) & \xrightarrow{\{\!|t'|\!\}} (t, E) & \end{array}$$

The execution of the branching threads at the end of the instant is defined as follows:

$$\begin{array}{ll} \lfloor 0 \rfloor_E = 0 & \lfloor \mathsf{present}\ s\ t\ b \rfloor_E = \langle\!| b |\!\rangle_E \\ \langle\!| t |\!\rangle_E = t & \langle\!| \mathsf{ite}\ s\ b_1\ b_2 |\!\rangle_E = \begin{cases} \langle\!| b_1 |\!\rangle_E & \text{if } E(s) = \mathit{true} \\ \langle\!| b_2 |\!\rangle_E & \text{if } E(s) = \mathit{false} \end{cases} \end{array}$$

A program is now a finite non-empty multi-set of tail recursive threads and program reduction is defined as in section 2.4. We can define the instructions pause and await in 'prefix form' as follows:

$$\begin{array}{ll} \mathsf{pause}.b & = \nu s\ \mathsf{present}\ s\ 0\ b \\ \mathsf{await}\ s.t & = A, \quad \text{where: } A = \mathsf{present}\ s\ t\ A, \quad \{\mathbf{s}\} = \mathit{sig}(t) \cup \{s\}\ . \end{array}$$

Determinism is guaranteed by the design of the language while reactivity can be enforced by a static analysis similar (but simpler) than the one presented in section 3.

4.1 CPS translation

We denote with ϵ an empty sequence. The translation $[\![\_]\!]$ described in table 2 has 2 parameters: (1) a thread t which stands for the *default continuation* and (2) a sequence $\tau \equiv (s_1, t_1) \cdots (s_n, t_n)$. If s_i is the 'first' (from left to right) signal which is present then t_i is the continuation. Whenever we cross a watch statement we insert a pair (s, t) in the sequence τ. Then we can translate the await statement with the present statement provided that at the end of each instant we check (from left to right) whether there is a pair (s, t) in τ such that the signal s is present. In this case, the continuation t must be run at the following instant.

Some later versions of the SL language include a (when s T) statement whose informal semantics is to run T (possibly over several instants) when s is present. It is possible to elaborate the CPS translation to handle this operator. The idea is to introduce as an additional parameter to the translation, the list of signals that have to be present for the computation to progress.

In the translation of a thread identifier, say, $A^{(t,\tau)}(\mathbf{x}, \mathbf{s}') = [\![T]\!](t, \tau)$ the identifier $A^{(t,\tau)}$ takes as additional parameters the signal names free in (t, τ). For the sake of readability, in the following we will simply write $A^{(t,\tau)}(\mathbf{x})$ and omit the parameters $\mathbf{s}'$.

It is important to notice that the translation associates with an equation $A(\mathbf{x}) = T$ a potentially infinite family of equations $A^{(t,\tau)}(\mathbf{x}) = [\![T]\!](t, \tau)$, the index (t, τ) depending on the evaluation context. However, whenever the evaluation contexts are 'bounded' in the sense described in the following section 4.2, only a finite number of indices are needed and the CPS translation preserves the finiteness of the system of recursive equations.

Example 8 *We compute the CPS translation of the thread A in example 7 (without unfolding). To keep the translation compact, we will use a slightly optimised CPS translation of the* pause *statement that goes as follows:*

$$[\![\mathsf{pause}]\!](t, (s_1, t_1) \cdots (s_n, t_n)) = \mathsf{pause}.\mathsf{ite}\ s_1\ t_1(\cdots(\mathsf{ite}\ s_n\ t_n\ t)\cdots)$$

Then the translation can be written as follows:

$$\begin{array}{llll} A^{(0,\epsilon)} & = B^{(t_1,\tau_1)} & t_1 & = \mathsf{emit}\ s_4.A^{(0,\epsilon)} \\ \tau_1 & = (s_1, t_1) & B^{(t_1,\tau_1)} & = \mathsf{present}\ s_2\ t_2\ (\mathsf{ite}\ s_1\ t_1\ B^{(t_1,\tau_1)}) \\ t_2 & = \mathsf{emit}\ s_3.\mathsf{pause}.\mathsf{ite}\ s_1\ t_1\ B^{(t_1,\tau_1)}\ . & & \end{array}$$

The translation is lifted to programs as follows: $[\![P]\!] = \{\![[\![T]\!](0, \epsilon) \mid T \in P]\!\}$. We show that a program generates exactly the same traces (cf. section 2.5) as its CPS translation. To this end, it is convenient to extend the CPS translation to evaluation contexts as follows:

$$\begin{array}{ll} [\![[\ ]]\!](t, \tau) & = (t, \tau) \\ [\![[\ ]; T]\!](t, \tau) & = ([\![T]\!](t, \tau), \tau) \\ [\![\mathsf{watch}\ s\ C]\!](t, \tau) & = [\![C]\!](t, \tau \cdot (s, t)) \\ [\![(\mathsf{watch}\ s\ C); T]\!](t, \tau) & = [\![C]\!]([\![T]\!](t, \tau), \tau \cdot (s, [\![T]\!](t, \tau))) \end{array}$$

Then we note the following decomposition property of the CPS translation whose proof is by induction on the evaluation context.

$$
\begin{array}{ll}
[\![0]\!](t,\tau) & = t \\
[\![T_1;T_2]\!](t,\tau) & = [\![T_1]\!]([\![T_2]\!](t,\tau),\tau) \\
[\![\mathsf{emit}\ s]\!](t,\tau) & = \mathsf{emit}\ s.t \\
[\![\nu s\ T]\!](t,\tau) & = \nu s\ [\![T]\!](t,\tau), \quad \text{where: } s \notin sig(t) \cup sig(\tau) \\
[\![\mathsf{thread}\ T]\!](t,\tau) & = \mathsf{thread}\ [\![T]\!](0,\epsilon).t \\
[\![\mathsf{watch}\ s\ T]\!](t,\tau) & = [\![T]\!](t,\tau\cdot(s,t)) \\
[\![\mathsf{await}\ s]\!](t,\tau) & = \mathsf{present}\ s\ t\ b, \quad \text{where: } \tau = (s_1,t_1)\cdots(s_m,t_m), \\
 & \quad b \equiv (\mathsf{ite}\ s_1\ t_1 \ldots (\mathsf{ite}\ s_m\ t_m\ A)\ldots), \quad A = \mathsf{present}\ s\ t\ b \\
[\![A(\mathbf{s})]\!](t,\tau) & = A^{(t,\tau)}(\mathbf{s},\mathbf{s}'), \quad \text{where: } sig(t,\tau) = \{\mathbf{s}'\}, \quad A(\mathbf{x}) = T, \\
 & \qquad\qquad\qquad\qquad \{\mathbf{x}\} \cap \{\mathbf{s}'\} = \emptyset, \quad A^{(t,\tau)}(\mathbf{x},\mathbf{s}') = [\![T]\!](t,\tau)\ .
\end{array}
$$

Table 2: A CPS translation

Proposition 9 *For all C evaluation context, T thread, t tail recursive thread, τ sequence,*

$$[\![C[T]]\!](t,\tau) = [\![T]\!]([\![C]\!](t,\tau))\ .$$

Definition 10 *We define a relation $\mathcal{R}$ between threads in the source and target language: $T\ \mathcal{R}\ t$ if either (1) $t = [\![T]\!](0,\epsilon)$ or (2) $T = C[\mathsf{await}\ s]$, $t = A$, and $A = [\![T]\!](0,\epsilon)$.*

The idea is that $T\ \mathcal{R}\ t$ if $t = [\![T]\!](0,\epsilon)$ up to the unfolding of the recursive definition in the CPS translation of an await statement. The need for the unfolding arises when checking the commutation of the CPS translation with the computation at the end of the instant. Then, we show that the relation $\mathcal{R}$ behaves as a kind of weak bisimulation with respect to reduction and suspension and that it is preserved by the computation at the end of the instant. This point requires a series of technical lemmas which are presented in appendix A.3. In turn, these lemmas entail directly the following theorem 11.

Theorem 11 *Let P be a program. Then $tr(P) = tr([\![P]\!])$.*

4.2 A static analysis to bound evaluation contexts

The source language allows an unlimited accumulation of evaluation contexts. To avoid stack overflow at run time, we define a simple control flow analysis that guarantees that each thread has an evaluation context of bounded size. For instance, have this property: (i) the fragment of the language using loop rather than recursive definitions and (ii) programs where recursive calls under a watch are guarded by a thread statement such as $A = (\mathsf{watch}\ s\ \mathsf{pause}; (\mathsf{thread}\ A))$. On the other hand, fail this property recursive definitions such as: (i) $A = \mathsf{pause}; A; B$ and (ii) $A = (\mathsf{watch}\ s\ \mathsf{pause}; A)$.

Let $L = \{\epsilon, \kappa\}$ be a set of labels. Intuitively, ϵ indicates an empty evaluation context, while κ indicates a (potentially) non-empty evaluation context. Sequential composition and the watch statement increase the size of the evaluation context while the thread statement resets its size to 0. Following this intuition, we define a function *Call* that associates with a thread and a label a set of pairs of thread identifiers and labels.

$$Call(0,\ell) = Call(\mathsf{await}\ s,\ell) = Call(\mathsf{emit}\ s,\ell) = \emptyset, \qquad Call(A,\ell) = \{(A,\ell)\},$$
$$Call(\mathsf{thread}\ T,\ell) = Call(T,\epsilon), \qquad Call(T_1;T_2,\ell) = Call(T_1,\kappa) \cup Call(T_2,\ell),$$
$$Call(\mathsf{watch}\ s\ T,\ell) = Call(T,\kappa)\ .$$

Definition 12 (constraints) *We denote with $Cnst(P)$ the least set of inequality and equality constraints on thread identifiers such that for any equation $A(\mathbf{x}) = T$ in the program P: (1) if $(B,\kappa) \in Call(T)$ then $A > B \in Cnst(P)$ and (2) if $(B,\epsilon) \in Call(T)$ then $A \geq B \in Cnst(P)$.*

If $\succeq$ is a pre-order we define: (i) $x \simeq y$ if $x \succeq y$ and $y \succeq x$ and (ii) $x \succ y$ if $x \succeq y$ and $x \not\simeq y$.

Definition 13 (satisfaction) *We say that a pre-order $\succeq$ on thread identifiers satisfies the constraints $Cnst(P)$ if: (1) $A > B \in Cnst(P)$ implies $A \succ B$, (2) $A \geq B \in Cnst(P)$ implies $A \succeq B$, and (3) $\succ$ is well-founded.*

We can now state the correctness of our criteria whose proof is delayed to appendix A.4. The reader may check the criteria on example 8.

Proposition 14 *If there is a pre-order that satisfies $Cnst(P)$ then the CPS translation preserves the finiteness of the system of equations.*

5 Expressivity

In this section we present two basic results on the computational expressivity of the model. First, we show that reactive programs without signal generation are trace equivalent to *monotonic* deterministic finite state machines, modulo a natural encoding. Second, we notice that the combination of recursion and signal name generation allows to simulate the computation of two counter machines. Thus, unlike the original SL language, it is not always possible to compile our programs to finite state machines.

5.1 Monotonic Mealy machines

A *monotonic* Mealy machine is a particular Mealy machine whose input and output alphabets are powersets and such that the function that determines the output respects the inclusion order on powersets. As for programs, we can associate with a monotonic Mealy machine a set of traces.

Definition 15 (monotonic Mealy machine) *A finite state, deterministic, reactive, and monotonic Mealy machine (monotonic Mealy machine for short) is a tuple $M = (Q, q_o, I, O, f_Q, f_O)$ where Q is a finite set of states, $q_o \in Q$ is the initial state, $I = 2^n$, $O = 2^m$ for n, m natural numbers are the input and output alphabets, respectively, $f_Q : I \times Q \to Q$ is the function computing the next state, and $f_O : I \times Q \to O$ is the function computing the output which is monotonic in the input, namely $X \subseteq Y$ implies $f_O(X, q) \subseteq f_O(Y, q)$.*

Theorem 16 *For every monotonic Mealy machine with input alphabet $I = 2^n$ and output alphabet $O = 2^m$ there is a trace equivalent program with n input signals and m output signals.*

Proof. The function $f_Q(\_, q)$ that for a given state q computes the next state as a function of the input can be coded as a cascade of ite's. The function $f_O(\_, q)$ that for a given state q computes the output as a function of the input can be coded as the parallel composition of threads that emit a certain output signal if a certain number of input signals is present in the instant and do nothing otherwise.

Next we develop some details. Let $M = (Q, q_o, I, O, f_Q, f_O)$ with $I = 2^n$ and $O = 2^m$ be a monotonic Mealy machine. We build the corresponding program. We introduce signals $s_1, \ldots, s_n$ for the input and signals $s'_1, \ldots, s'_m$ for the output. Moreover, we introduce a thread identifier q for every state $q \in Q$. Given a state q, we associate with the function $f_Q(\_, q) : 2^n \to Q$ a branching thread $b(q)$. For instance, if the function is defined by:

$$f_Q((1,1), q) = q_1, \quad f_Q((1,0), q) = q_2, \quad f_Q((0,1), q) = q_3, \quad f_Q((0,0), q) = q_1,$$

then the corresponding branching thread is:

$$b(q) = \mathsf{ite}\ s_1\ (\mathsf{ite}\ s_2\ q_1\ q_2)\ (\mathsf{ite}\ s_2\ q_3\ q_1)$$

For every state q, we introduce an equation of the shape:

$$q = \mathit{Output}(q).\mathsf{pause}.b(q) \tag{2}$$

where $\mathit{Output}(q)$ is intended to compute the output function $f_O(\_, q) : 2^n \to 2^m$. To formalise this, we need some notation. Let $X \subseteq \{1, \ldots, n\}$ denote an input symbol and $j \in \{1, \ldots, m\}$. By monotonicity, if $X \subseteq Y$ and $j \in f_O(X, q)$ then $j \in f_O(Y, q)$. Given a family of threads $\{t_j\}_{j \in J}$, we write $\mathsf{thread}_{j \in J} t_j.t$ for the thread that spawns, in an arbitrary order, the threads t_j and then runs t. Given a set of input signals $\{s_1, \ldots, s_k\}$ and an output signal s'_j, we write $\mathsf{await}\{s_1, \ldots, s_k\}.t$ for

$$\mathsf{present}\ s_1\ (\cdots (\mathsf{present}\ s_k\ t\ 0) \cdots)\ 0$$

which executes t in the first instant it is run if and only if all the signals $s_1, \ldots, s_k$ are present, and terminates otherwise. No signals are emitted in the instants following the first one. With these conventions $\mathit{Output}(q).t$ is an abbreviation for

$$(\ \mathsf{thread}_{X \subseteq \{1,\ldots,n\},\ j \in f_O(X,q)}\ (\mathsf{await}\ \{s_x \mid x \in X\}.\ \mathsf{emit}\ s'_j)\ ).\ t$$

so that the explicit form for equation (2) is:

$$q = (\ \mathsf{thread}_{X\subseteq\{1,\ldots,n\},\ j\in f_O(X,q)}\ (\mathsf{await}\ \{s_x \mid x \in X\}.\ \mathsf{emit}\ s'_j)\ ).\ \mathsf{pause}.\ b(q)\ .$$

$\Box$

One may wonder whether our synchronous language may represent *non-monotonic* Mealy machines. The answer to this question is negative as long we adopt the encoding of the input above where 2^n input symbols are mapped to n signals. This fact easily follows from the monotonicity property of the model noted in section 3. However, the answer is positive if we adopt a less compact representation where n input symbols are mapped to n signals.

Next we focus on the expressive power of the reactive programs we can write in the tail recursive calculus presented in section 4 *without signal generation* but with general recursion and thread spawning.

Theorem 17 *For every reactive tail recursive program with n input signals and m output signals and without signal generation there is a trace equivalent monotonic Mealy machine with input alphabet 2^n and output alphabet 2^m.*

PROOF. The construction takes several steps but the basic idea is simple: it is useless to run twice or more times through the same 'control point' within the same instant. Instead we record the set of control points that have been reached along with the signals that have been emitted and in doing so we are bound to reach a fixed point.

We start with some preliminary considerations that allow to simplify the representation of programs.

(1) Since there is no signal generation a program depends on a finite set S_o of signal names. As a first step we can remove parameters from recursive equations. To this end, replace every parametric equation $A(\mathbf{x}) = t$ with a finite number of equations (without parameters) of the shape $A_{\mathbf{s}} = [\mathbf{s}/\mathbf{x}]t$ for $\mathbf{s}$ ranging over tuples of signal names in S_o.

(2) As a second step, we put the recursive equations in normal form. By introducing auxiliary thread identifiers, we may assume the equations have the shape $A = t$ where

$$\begin{array}{lcl} t & ::= & 0 \mid \mathsf{emit}\ s.B \mid \mathsf{present}\ s\ B\ b \mid \mathsf{thread}\ B.B' \\ b & ::= & A \mid \mathsf{ite}\ s\ b\ b \end{array}$$

We denote with Id_o the finite set of thread identifiers.

(3) Because there is no signal name generation, we may simply represent the environment E as a subset of S_o and because the threads are in normal form we may simply represent a program P as a multi-set of identifiers in Id_o. The small step reduction of the pair (P, E) is then described as follows:

$$(P \cup \{\!|A|\!\}, E) \rightarrow \left\{ \begin{array}{ll} (P \cup \{\!|B|\!\}, E \cup \{s\}) & \text{if } A = \mathsf{emit}\ s.B \\ (P \cup \{\!|B|\!\}, E) & \text{if } A = \mathsf{present}\ s\ B\ b,\ \ s \in E \\ (P \cup \{\!|B_1, B_2|\!\}, E) & \text{if } A = \mathsf{thread}\ B_1.B_2 \end{array} \right.$$

Notice that in this presentation, the unfolding of recursive definitions is kept implicit. If the program is reactive we know that the evaluation of a pair (P, E) eventually terminates in a configuration (P', E') such that if $A \in P'$ then either $A = 0$ or $A = \mathsf{present}\ s\ B\ b$ and $s \notin E'$. The evaluation at the end of the instant $\lfloor P' \rfloor_{E'}$ is then a particular case of the one defined in section 4 for tail recursive threads and produces again a multi-set of thread identifiers.

(4) We now consider an alternative representation of a program as a *set* q of identifiers in Id_o. We define a small step reduction on configurations (q, E) as follows:

$$(q \cup \{A\}, E) \rightarrow \begin{cases} (q \cup \{A, B\}, E \cup \{s\}) & \text{if } A = \mathsf{emit}\ s.B,\ (B \notin q \cup \{A\} \text{ or } s \notin E) \\ (q \cup \{A, B\}, E) & \text{if } A = \mathsf{present}\ s\ B\ b,\ s \in E,\ B \notin q \cup \{A\} \\ (q \cup \{A, B_1, B_2\}, E) & \text{if } A = \mathsf{thread}\ B_1.B_2,\ \{B_1, B_2\} \not\subseteq q \cup \{A\} \end{cases}$$

Note that at each reduction step either the program q or the environment E increase strictly while the other component does not decrease. Consequently, this reduction process (unlike the previous one) necessarily terminates. The evaluation at the end of the instant is now defined as follows:

$$\lfloor q \rfloor_E = \{A \in q \mid A = 0\} \cup \{\langle\!| b |\!\rangle_E \mid A \in q, A = \mathsf{present}\ s\ B\ b, \text{ and } s \notin E\}\ .$$

Notice that q may contain, *e.g.*, a thread identifier A such as $A = \mathsf{emit}\ s.B$ and that A is removed by the function $\lfloor \_ \rfloor_E$.

(5) We now relate the two representations of the programs and the associated evaluation strategies where if P is a multi-set we let $set(P) = \{A \mid A \in P\}$ be the corresponding set where we forget multiplicities.

Lemma 18 *Suppose* $(P_1, E_1) \rightarrow \cdots \rightarrow (P_n, E_n)$ *with* $n \geq 1$ *and* $q = set(P_1 \cup \cdots \cup P_n)$. *Then:*
(1) *If* $(P_n, E_n) \rightarrow (P_{n+1}, E_{n+1})$ *then either* $E_n = E_{n+1}$ *and* $set(P_{n+1}) \subseteq q$ *or* $(q, E_n) \rightarrow (q', E_{n+1})$ *and* $q' = set(P_1 \cup \cdots \cup P_{n+1})$.
(2) *If* $(q, E_n) \rightarrow (q', E_{n+1})$ *then* $(P_n, E_n) \rightarrow (P_{n+1}, E_{n+1})$ *and* $q' = set(P_1 \cup \cdots \cup P_{n+1})$.
(3) *If* $(P_n, E_n) \downarrow$ *then* $set(\lfloor P_n \rfloor_{E_n}) = \lfloor q \rfloor_{E_n}$.

PROOF. (1) By case analysis on the small step reduction for multi-sets.

(2) By case analysis on the small step reduction for sets. Note that if the reduction rule is applied to $A \in q$ then necessarily $A \in P_n$. Indeed, if $A \in P_k$ and $A \notin P_{k+1}$ with $k < n$ we can conclude that a reduction rule has been applied to A on the multi-set side and this contradicts the hypotheses for the firing of the rule on the set side.

(3) We check that if $A = 0$ and $A \in q$ then $A \in P_n$ and that if $A = \mathsf{present}\ s\ B\ t$, $s \notin E_n$ and $A \in q$ then $A \in P_n$. $\Box$

(6) We define

$$Closure(q, E) = (q', E') \text{ if } (q, E) \rightarrow \cdots \rightarrow (q'', E') \not\rightarrow \text{ and } q' = \lfloor q'' \rfloor_{E'}$$

The *Closure* operator is well defined because the reduction relation is strongly confluent and it always terminates.

(7) As a final step, given a reactive program P in normal form with identifiers Id_o, n input signals $s_1, \ldots, s_n$ and m output signals $s'_1, \ldots, s'_m$, we build a trace equivalent monotonic Mealy machine $M = (Q, q_o, I, O, f_Q, f_O)$ as follows: $Q = 2^{Id_o}$, $q_o = set(P)$, $I = 2^n$, $O = 2^m$, and $(f_Q(E,q), f_O(E,q)) = Closure(q, E)$. □

By combining theorems 16 and 17, we can conclude that the reactive programs we can write without signal generation are exactly those definable by monotonic Mealy machines modulo a natural encoding.

5.2 Undecidability

The following result can be used to show that various questions about the behaviours of programs are undecidable. The encoding idea is similar to the one presented for CCS in [15]. The details are presented in appendix A.5.

Theorem 19 *For any deterministic 2-counter machine there is a reactive program with signal generation that will eventually emit on a certain signal if and only if the computation of the 2-counter machine terminates.*

6 Program equivalence

The formalisation of the SL model we have considered so far is close to an abstract machine. Typical symptoms include an *ad hoc* definition of α-renaming (cf. section 3), a global notion of environment, and the fact that roughly threads compose but do not reduce while programs reduce but do not compose. We introduce next an alternative description of the tail recursive model featuring a uniform notation for threads, programs, and environments. This alternative description is instrumental to the development of a notion of program equivalence based on the concept of bisimulation following a CCS style. The theory is built so that it does not depend on the determinacy of the language. Indeed *practical* extensions of the language have been considered where signals carry data values and the act of receiving a value may introduce non-determinism. A theory of program equivalence should be sufficiently robust to accommodate these extensions.

6.1 Programs

We extend the syntax of tail recursive threads so that it includes both environments and programs in a uniform notation.

$$\begin{array}{ll} P & ::= 0 \mid \mathsf{emit}\ s \mid \mathsf{present}\ s\ P\ B \mid P \mid P \mid \nu s\ P \mid A(\mathbf{s}) \\ B & ::= P \mid \mathsf{ite}\ s\ B\ B \end{array}$$

We refrain from introducing syntax like '$\mathsf{emit}\ s.P$ and '$\mathsf{thread}\ P'.P$ which can be understood as syntactic sugar for $(\mathsf{emit}\ s) \mid P$ and $P' \mid P$, respectively.

6.2 Actions and labelled transition system

Actions are denoted by $\alpha, \alpha', \ldots$ and they are defined by the grammar: $\alpha ::= \tau \mid s \mid \overline{s}$. We write $s \in \alpha$ if $\alpha = s$ or $\alpha = \overline{s}$. We define a *labelled transition system* which is similar to the one for CCS except for a different treatment of emission which is *persistent* within an instant. Technically, (i) an emission behaves as a replicated output (rule (*out*)) and (ii) in the continuation of a $\mathsf{present}$ statement the tested signal is still emitted (rule (*in*)); this guarantees that the continuation can only evolve in an environment where the signal s is emitted.[1]

$$
\begin{array}{llll}
(out) & \dfrac{}{\mathsf{emit}\ s \xrightarrow{\overline{s}} \mathsf{emit}\ s} & (in) & \dfrac{}{\mathsf{present}\ s\ P\ B \xrightarrow{s} P \mid (\mathsf{emit}\ s)} \\
(\tau) & \dfrac{P_1 \xrightarrow{s} P_1' \quad P_2 \xrightarrow{\overline{s}} P_2'}{P_1 \mid P_2 \xrightarrow{\tau} P_1' \mid P_2'} & (par) & \dfrac{P_1 \xrightarrow{\alpha} P_1'}{P_1 \mid P_2 \xrightarrow{\alpha} P_1' \mid P_2} \\
(\nu) & \dfrac{P \xrightarrow{\alpha} P' \quad s \notin \alpha}{\nu s\ P \xrightarrow{\alpha} \nu s\ P'} & (rec) & \dfrac{A(\mathbf{x}) = P}{A(\mathbf{s}) \xrightarrow{\tau} [\mathbf{s}/\mathbf{x}]P}
\end{array}
$$

As usual, we omit the symmetric rules for (par, τ). We note the following properties of the labelled transition system where $=$ stands for syntactic identity up to renaming of bound names.

Proposition 20 (1) *If $P \xrightarrow{\overline{s}} P'$ then $P = P'$.*

(2) *If $P \xrightarrow{\overline{s}} P$ and $P \xrightarrow{\alpha} P'$ then $P' \xrightarrow{\overline{s}} P'$.*

(3) *If $P \xrightarrow{s} P'$ then $P' \xrightarrow{\overline{s}} P'$.*

6.3 End of the instant

We define the computation at the end of the instant while relying on the following notation: $P \xrightarrow{\alpha} \cdot$ for $\exists P'\ \ P \xrightarrow{\alpha} P'$ and $P \downarrow$ for $\neg(P \xrightarrow{\tau} \cdot)$. Suppose $P \downarrow$ and all bound signal names in P are renamed so as to be distinct and different from the free signal names. First, we compute the set of emitted signals $S = Em(P)$ as follows:

$$
Em(\mathsf{emit}\ s) = \{s\}, \quad Em(0) = Em(\mathsf{present}\ s\ P\ B) = \emptyset,
$$
$$
Em(P_1 \mid P_2) = Em(P_1) \cup Em(P_2), \quad Em(\nu s\ P) = Em(P)\ .
$$

[1]This is close in spirit, if not in the technical development, to Prasad's Calculus of Broadcasting Systems [18]; see also [10].

Second, we compute $\lfloor P \rfloor = \lfloor P \rfloor_{Em(P)}$ where we remove all emitted signals and compute the B branches relying on the auxiliary functions $\lfloor \_ \rfloor_S$ and $\langle\!| \_ |\!\rangle_S$ defined as follows:

$$\begin{array}{l}
\lfloor \mathsf{emit}\ s \rfloor_S = \lfloor 0 \rfloor_S = 0, \quad \lfloor \mathsf{present}\ s\ P\ B \rfloor_S = \langle\!| B |\!\rangle_S, \\
\lfloor \nu s\ P \rfloor_S = \nu s\ \lfloor P \rfloor_S, \quad \lfloor P_1 \mid P_2 \rfloor_S = \lfloor P_1 \rfloor_S \mid \lfloor P_2 \rfloor_S, \\
\langle\!| P |\!\rangle_S = P, \quad \langle\!| \mathsf{ite}\ s\ B_1\ B_2 |\!\rangle_S = \begin{cases} \langle\!| B_1 |\!\rangle_S & \text{if } s \in S \\ \langle\!| B_2 |\!\rangle_S & \text{if } s \notin S\ . \end{cases}
\end{array}$$

One can verify that the function $\lfloor \_ \rfloor$ is invariant under α-renaming: if $P_1 = P_2$ then $\lfloor P_1 \rfloor = \lfloor P_2 \rfloor$.

6.4 Barbed and contextual bisimulations

As usual, we write $P \overset{\tau}{\Rightarrow} P'$ for $P(\xrightarrow{\tau})^* P'$ and $P \overset{\alpha}{\Rightarrow} P'$ with $\alpha \neq \tau$ for $P(\overset{\tau}{\Rightarrow})(\xrightarrow{\alpha})(\overset{\tau}{\Rightarrow})P'$.

Definition 21 *We define:*

$$\begin{array}{lll}
P \Downarrow & \textit{if } \exists P'\ \ P \overset{\tau}{\Rightarrow} P' \textit{ and } P' \downarrow & \textit{(weak suspension)} \\
P \Downarrow_L & \textit{if } P \xrightarrow{\alpha_1} P_1 \cdots \xrightarrow{\alpha_n} P_n, \quad n \geq 0, \textit{ and } P_n \downarrow & \textit{(L-suspension)}
\end{array}$$

Obviously $P \downarrow$ implies $P \Downarrow$ which in turn implies $P \Downarrow_L$. The L-suspension predicate (L for labelled) plays an important role in the following definitions of bisimulation.

Definition 22 *A (static) context C is defined by $C ::= [\ ] \,\|\, C \mid P \,\|\, \nu s\ C$.*

Proposition 23 *Let P be a program. The following are equivalent:*

(1) $P \Downarrow_L$.

(2) *There is a program Q such that $(P \mid Q) \Downarrow$.*

(3) *There is a static context C such that $C[P] \Downarrow_L$.*

PROOF. $(1 \Rightarrow 2)$ Suppose $P_0 \xrightarrow{\alpha_1} P_1 \cdots \xrightarrow{\alpha_n} P_n$ and $P_n \downarrow$. We build Q by induction on n. If $n = 0$ we take $Q = 0$. Otherwise, suppose $n > 0$. By inductive hypothesis, there is Q_1 such that $(P_1 \mid Q_1) \Downarrow$. We proceed by case analysis on the first action α_1. We may assume α_1 is not an emission action for otherwise we can build a shorter sequence of transitions.

$(\alpha_1 = \tau)$ Then we take $Q = Q_1$ and $(P_0 \mid Q_1) \xrightarrow{\tau} (P_1 \mid Q_1)$.

$(\alpha_1 = s)$ Let $Q = (Q_1 \mid \overline{s})$. We have $(P_0 \mid Q) \xrightarrow{\tau} (P_1 \mid Q_1 \mid \overline{s})$. Since $P_1 \xrightarrow{\overline{s}} P_1$, we observe that $(P_1 \mid Q_1) \Downarrow$ implies $(P_1 \mid Q_1 \mid \overline{s}) \Downarrow$.

$(2 \Rightarrow 3)$ Take $C = [\ ] \mid Q$.

$(3 \Rightarrow 1)$ First, check by induction on a static context C that $P \xrightarrow{\tau} \cdot$ implies $C[P] \xrightarrow{\tau} \cdot$. Hence $C[P] \downarrow$ implies $P \downarrow$. Second, show that $C[P] \xrightarrow{\alpha} Q$ implies that $Q = C'[P']$ and either $P = P'$ or $P \xrightarrow{\alpha'} P$. Third, suppose $C[P] \xrightarrow{\alpha_1} Q_1 \cdots \xrightarrow{\alpha_n} Q_n$ with $Q_n \downarrow$. Show by induction on n that $P \Downarrow_L$. Proceed by case analysis on the context C and the action α_1. $\Box$

Interestingly, the second characterisation, shows that the L-suspension predicate can be defined just in terms of the τ transitions and the suspension predicate. This means that the following definitions of barbed and contextual bisimulation can be given *independently* of the labelled transition system.

Definition 24 (barbed bisimulation) *A symmetric relation R on programs is a barbed bisimulation if whenever $P \; R \; Q$ the following holds:*
$(B1)$ *If $P \xrightarrow{\tau} P'$ then $\exists \, Q' \;\; Q \overset{\tau}{\Rightarrow} Q'$ and $P' \; R \; Q'$.*
$(B2)$ *If $P \downarrow$ then $\exists \, Q' \;\; Q \overset{\tau}{\Rightarrow} Q', Q' \downarrow, P \; R \; Q'$, and $\lfloor P \rfloor \; R \; \lfloor Q' \rfloor$.*
$(B3)$ *If $P \xrightarrow{\overline{s}} \cdot$ and $P \Downarrow_L$ then $\exists \, Q' \;\; Q \overset{\tau}{\Rightarrow} Q', Q' \xrightarrow{\overline{s}} \cdot$, and $P \; R \; Q'$.*
We denote with $\approx_B$ the largest barbed bisimulation.

It is easily checked that $\approx_B$ is reflexive and transitive. A reasonable notion of program equivalence should be preserved by the static contexts. We define accordingly a notion of contextual bisimulation.[2]

Definition 25 (contextual bisimulation) *A symmetric relation R on programs is a contextual bisimulation if it is a barbed bisimulation (conditions B1-3) and moreover whenever $P \; R \; Q$ then*
$(C1)$ *$C[P] \; R \; C[Q]$, for any context C.*
We denote with $\approx_C$ the largest contextual bisimulation.

Again it is easily checked that $\approx_C$ is reflexive and transitive. By its very definition, it follows that $P \approx_C Q$ implies $C[P] \approx_C C[Q]$ and $P \approx_B Q$.

6.5 Labelled bisimulation

Aiming at a more effective description of the notion of contextual bisimulation, we introduce a notion of *labelled* bisimulation.

Definition 26 (labelled bisimulation) *A symmetric relation R on programs is a labelled bisimulation if it is a barbed bisimulation (conditions B1-3) and moreover whenever $P \; R \; Q$ the following holds:*
$(L1)$ *If $P' = (P \mid S) \downarrow$ with $S = \mathsf{emit}\ s_1 \mid \cdots \mid \mathsf{emit}\ s_n$, $n \geq 0$ then $\exists \, Q' \;\; (Q \mid S) \overset{\tau}{\Rightarrow} Q'$, $Q' \downarrow$, $P' \; R \; Q'$, and $\lfloor P' \rfloor \; R \; \lfloor Q' \rfloor$.*
$(L2)$ *If $P \xrightarrow{s} P'$ then either $\exists \, Q' \; (\; Q \overset{s}{\Rightarrow} Q'$ and $P' \; R \; Q')$ or $\exists \, Q' \; (\; Q \overset{\tau}{\Rightarrow} Q'$ and $P' \; R \; (Q' \mid \mathsf{emit}\ s) \;)$.*
We denote with $\approx_L$ the largest labelled bisimulation.

[2]Here we adopt the notion of contextual equivalence introduced by [11] for the π-calculus. An alternative approach is to consider a notion of *barbed equivalence* [16]. We refer to [9] for a comparison of the two methods.

Remark 27 (1) *Condition* $(L1)$ *strengthens* $(B2)$ *therefore in the following proof the analysis of* $(B2)$ *is subsumed by the one of* $(L1)$*. To see the necessity of condition* $(L1)$*, consider*

$$P = \mathsf{present}\ s_1\ 0\ (\mathsf{ite}\ s_2\ (\mathsf{emit}\ s_3)\ 0) \quad \textit{and} \quad Q = \mathsf{present}\ s_2\ 0\ 0\ .$$

Then $P \downarrow$*,* $Q \downarrow$*, and* $\lfloor P \rfloor = \lfloor Q \rfloor = 0$ *so that conditions* $(B1-3)$ *and* $(L2)$ *are satisfied. However, if we plug* P *and* Q *in the context* $[\,] \mid (\mathsf{emit}\ s_2)$ *then the resulting programs exhibit different behaviours. It is not difficult to show that condition* $(L1)$ *can be optimised so that we only consider emissions on signals which are free in the programs under consideration. For instance, a simple corollary of this optimisation is that labelled bisimulation is decidable for programs* without *recursive definitions.*

(2) *Condition* $(L2)$ *has already appeared in the literature in the context of the asynchronous* π*-calculus [2].*

(3) *There is no condition for the emission because by proposition 20 condition* $(B3)$ *is equivalent to the following one: if* $P \xrightarrow{\overline{s}} P'$ *and* $P' \Downarrow_L$ *then* $\exists\, Q'$ $(\ Q \overset{\overline{s}}{\Rightarrow} Q'$ *and* $P'\ R\ Q'\ )$*.*

(4) *The condition* $P \Downarrow_L$ *in* $(B3)$ *is always satisfied by reactive programs which are those we are really interested in. We will see in section 6.9, that thanks to strong confluence, the condition* $P \Downarrow_L$ *can be replaced by the condition* $P \Downarrow$ *or equivalently by the condition* $P \downarrow$*. However, one should keep in mind that there are non-deterministic extensions of the language where this identification fails and where moreover the definitions based on the weaker conditions* $P \downarrow$ *or* $P \Downarrow$ *lead to notions of labelled bisimulation which are not preserved by parallel composition. For this reason, our definitions of bisimulation are based on the L-suspension predicate.*

We can now state the main result of this section.

Theorem 28 $P \approx_C Q$ *iff* $P \approx_L Q$*.*

We outline the proof argument which is developed in the following. First, we note that labelled bisimulation equates all programs which cannot L-suspend and moreover it never equates a program which L-suspends to one which cannot. Second, we introduce a notion of *strong* labelled bisimulation which is contained in labelled bisimulation. It is shown that strong labelled bisimulation satisfies some useful laws like associativity, commutativity, commutation of signal name generation, ... Third, we develop a notion of labelled bisimulation up to strong labelled bisimulation that considerably simplifies reasoning about labelled bisimulation. Fourth, we show that $\approx_C$ is a labelled bisimulation up to strong labelled bisimulation so that $P \approx_C Q$ implies $P \approx_L Q$. Fifth, we show that labelled bisimulation is preserved by parallel composition with signal emission, it is reflexive and transitive, and it is preserved by signal name generation, parallel composition, and the present operator. In particular, it follows that $\approx_L$ is preserved by the static contexts, *i.e.*, $\approx_L$ is a contextual barbed bisimulation and therefore $P \approx_L Q$ implies $P \approx_C Q$.

6.6 Labelled bisimulation and L-suspension

We observe some remarkable properties of the L-suspension predicate.

Proposition 29 (1) *If* $\neg P \Downarrow_L$ *and* $\neg Q \Downarrow_L$ *then* $P \approx_L Q$.
(2) *If* $P \approx_L Q$ *and* $P \Downarrow_L$ *then* $Q \Downarrow_L$.

Proof. First we note the following properties:
(A) By proposition 23, if $(P \mid Q) \Downarrow_L$ then $P \Downarrow_L$.
(B) By definition, if $\neg P \Downarrow_L$ and $P \xrightarrow{\alpha} P'$ then $\neg P' \Downarrow_L$.

(1) We show that $\{(P,Q) \mid \neg P \Downarrow_L \text{ and } \neg Q \Downarrow_L\}$ is a labelled bisimulation.
$(B1)$ By (B), if $\neg P \Downarrow_L$ and $P \xrightarrow{\tau} P'$ then $\neg P' \Downarrow_L$.
$(B3)$ The hypothesis is not satisfied.
$(L1)$ By (A), if $\neg P \Downarrow_L$ then $\neg(P \mid S) \Downarrow_L$. Hence $\neg(P \mid S) \downarrow$.
$(L2)$ By (B), if $\neg P \Downarrow_L$ and $P \xrightarrow{s} P'$ then $\neg P' \Downarrow_L$. Then we match the transition with $Q \overset{\tau}{\Rightarrow} Q$ and by (A) $\neg Q \Downarrow_L$ implies $\neg(Q \mid (\mathsf{emit}\ s)) \Downarrow_L$.

(2) We proceed by induction on the shortest reduction such that $P \xrightarrow{\alpha_1} P_1 \cdots \xrightarrow{\alpha_n} P_n$ and $P_n \downarrow$. Note that in such a reduction no emission action $\overline{s}$ occurs (otherwise a shortest reduction can be found). If $n = 0$ then $(B2)$ requires $Q \overset{\tau}{\Rightarrow} Q'$ and $Q' \downarrow$. Hence $Q \Downarrow_L$. If $n > 0$ then we consider the first action α_1. If $\alpha_1 = \tau$ then $(B1)$ requires $Q \overset{\tau}{\Rightarrow} Q_1$ and $P_1 \approx_L Q_1$. Then $Q_1 \Downarrow_L$ by inductive hypothesis on P_1. Hence $Q \Downarrow_L$. If $\alpha_1 = s$ then we have to consider two cases. If $Q \overset{s}{\Rightarrow} Q_1$ and $P_1 \approx_L Q_1$ then $Q_1 \Downarrow_L$ by inductive hypothesis on P_1. Hence $Q \Downarrow_L$. If on the other hand $Q \overset{\tau}{\Rightarrow} Q_1$ and $P_1 \approx_L Q_1 \mid (\mathsf{emit}\ s)$ then $Q_1 \mid (\mathsf{emit}\ s) \Downarrow_L$. Hence by (A) $Q_1 \Downarrow_L$, and $Q \Downarrow_L$. $\square$

6.7 Strong labelled bisimulation and an up-to technique

To bootstrap reasoning about labelled bisimulation, it is convenient to introduce a much stronger notion of labelled bisimulation.

Definition 30 (strong labelled bisimulation) *A symmetric relation* R *on programs is a strong labelled bisimulation if whenever* $P\ R\ Q$ *the following holds:*
$(S1)$ $P \xrightarrow{\alpha} P'$ *implies* $\exists Q'\ Q \xrightarrow{\alpha} Q'$ *and* $P'\ R\ Q'$.
$(S2)$ $(P \mid S) \downarrow$ *with* $S = (\mathsf{emit}\ s_1) \mid \cdots \mid (\mathsf{emit}\ s_n)$, $n \geq 0$ *implies* $(P \mid S)\ R\ (Q \mid S)$ *and* $\lfloor P \mid S \rfloor\ R\ \lfloor Q \mid S \rfloor$.[3]
We denote with $\equiv_L$ *the largest strong labelled bisimulation.*

[3] The condition $(Q \mid S) \downarrow$ follows by $(S1)$.

Note that in definition 30 not only we forbid weak internal moves but we also drop the convergence condition in $(B3)$ and the possibility of matching an input with an internal transition in $(L2)$. For this reason, we adopt the notation $\equiv_L$ rather than the usual $\sim_L$. We say that a relation R is a strong labelled bisimulation up to strong labelled bisimulation if the conditions $(S1-2)$ hold when we replace R with the larger relation $(\equiv_L) \circ R \circ (\equiv_L)$. Strong labelled bisimulation enjoys some useful properties whose standard proof is delayed to appendix A.7

Lemma 31 (1) *$\equiv_L$ is a reflexive and transitive relation.*

(2) *If $P \equiv_L Q$ then $P \approx_L Q$.*

(3) *The following laws hold:*

$$\begin{array}{ll} P \mid 0 \equiv_L P, & P_1 \mid (P_2 \mid P_3) \equiv_L (P_1 \mid P_2) \mid P_3, \\ P_1 \mid P_2 \equiv_L P_2 \mid P_1, & \nu s\ P_1 \mid P_2 \equiv_L \nu s\ (P_1 \mid P_2) \textit{ if } s \notin sig(P_2). \end{array}$$

(4) *If $P \equiv_L Q$ then $P \mid S \equiv_L Q \mid S$ where $S = P_1 \mid \cdots \mid P_n$ and $P_i = 0$ or $P_i = (\mathsf{emit}\ s_i)$, for $i = 1, \ldots, n$, $n \geq 0$.*

(5) *If R is a strong labelled bisimulation up to strong labelled bisimulation then $(\equiv_L) \circ R \circ (\equiv_L)$ is a strong labelled bisimulation.*

(6) *If $P \xrightarrow{\bar{s}} \cdot$ then $P \equiv_L P \mid (\mathsf{emit}\ s)$.*

(7) *If $P_1 \equiv_L P_2$, then $\nu s\ P_1 \equiv_L \nu s\ P_2$ and $P_1 \mid Q \equiv_L P_2 \mid Q$.*

We use strong labelled bisimulation in the context of a rather standard 'up to technique'.

Definition 32 *A relation R is a labelled bisimulation up to $\equiv_L$ if the conditions $(B1-3)$ and $(L1-2)$ are satisfied when replacing the relation R with the (larger) relation $(\equiv_L) \circ R \circ (\equiv_L)$.*

Lemma 33 *Let R be a labelled bisimulation up to $\equiv_L$. Then:*

(1) *The relation $(\equiv_L) \circ R \circ (\equiv_L)$ is a labelled bisimulation.*

(2) *If $P\ R\ Q$ then $P \approx_L Q$.*

Proof. (1) A direct diagram chasing using the congruence properties of $\equiv_L$.

(2) Follows directly from (1). □

6.8 Characterisation

As a first application of the 'up to technique', we show that $P \approx_C Q$ implies $P \approx_L Q$.

Lemma 34 *$\approx_C$ is a labelled bisimulation up to $\equiv_L$.*

PROOF. Suppose $P \approx_C Q$. We check conditions $(L1-2)$.

$(L1)$ Suppose $S = (\mathsf{emit}\ s_1) \mid \cdots \mid (\mathsf{emit}\ s_n)$ and $(P \mid S) \downarrow$. Since $\approx_C$ is preserved by parallel composition we derive $P \mid S \approx_C Q \mid S$. Then we conclude by applying condition $(B2)$.

$(L2)$ Suppose $P \xrightarrow{s} P'$. By lemma 31(6), this implies $P' \equiv_L P' \mid (\mathsf{emit}\ s)$. Since $\approx_C$ is preserved by parallel composition we know $P \mid (\mathsf{emit}\ s) \approx_C Q \mid (\mathsf{emit}\ s)$. From this and the fact that $P \mid (\mathsf{emit}\ s) \xrightarrow{\tau} P' \mid (\mathsf{emit}\ s)$ condition $(B1)$ allows to derive that $Q \mid (\mathsf{emit}\ s) \overset{\tau}{\Rightarrow} Q' \mid (\mathsf{emit}\ s)$ and $P' \mid (\mathsf{emit}\ s) \approx_C Q' \mid (\mathsf{emit}\ s)$. Two cases may arise: (1) $Q \overset{s}{\Rightarrow} Q'$. Then we have $P' \equiv_L P' \mid (\mathsf{emit}\ s) \approx_C Q' \mid (\mathsf{emit}\ s) \equiv_L Q'$. (2) $Q \overset{\tau}{\Rightarrow} Q'$. Then we have $P' \equiv_L P' \mid (\mathsf{emit}\ s) \approx_C Q' \mid (\mathsf{emit}\ s)$. In both cases we close the diagram up to $\equiv_L$. $\Box$

As a second application of the 'up to technique' we prove some desirable congruence properties of the labelled bisimulation (the proofs are delayed to appendix A.8). Assume $\mathsf{pause}.B$ abbreviates $\nu s\ \mathsf{present}\ s\ 0\ B$ for $s \notin sig(B)$. We write $B_1 \approx_L B_2$ if $\mathsf{pause}.B_1 \approx_L \mathsf{pause}.B_2$.

Lemma 35 (1) *If $P \approx_L Q$ then $P \mid (\mathsf{emit}\ s) \approx_L Q \mid (\mathsf{emit}\ s)$.*
(2) *The relation $\approx_L$ is reflexive and transitive.*
(3) *If $P \approx_L Q$ then $\nu s\ P \approx_L \nu s\ Q$.*
(4) *If $P_1 \approx_L P_2$ then $P_1 \mid Q \approx_L P_2 \mid Q$.*
(5) *If $P \approx_L P'$ and $B \approx_L B'$ then $\mathsf{present}\ s\ P\ B \approx_L \mathsf{present}\ s\ P'\ B'$.*

The lemma above entails that $\approx_L$ is preserved by static contexts. Hence $P \approx_L Q$ implies $P \approx_C Q$. This remark combined with lemma 34 concludes the proof of theorem 28.

6.9 Exploiting confluence

We can easily adapt the trace semantics presented in section 2.5 to the present context. If P is a program we write (Π for the parallel composition):

$$P \xrightarrow{I/O} P' \text{ if } P \mid P_I \overset{\tau}{\Rightarrow} P'',\ \text{with } P_I = \Pi_{s\in I}\overline{s},\quad P'' \downarrow,\quad O = \{s \mid P'' \xrightarrow{\overline{s}} \cdot\},\ \text{and } P' = \lfloor P'' \rfloor\ .$$

and we associate with P a set of traces $tr(P)$ as in section 2.5. A general argument shows that labelled bisimulation is a refinement of trace equivalence.

Proposition 36 *If $P \approx_L Q$ then $tr(P) = tr(Q)$.*

PROOF. We observe that if $P \approx_L Q$ and $P \xrightarrow{I/O} P'$ then $Q \xrightarrow{I/O} Q'$ and $P' \approx_L Q'$. From this one can show that every trace in $tr(P)$ is in $tr(Q)$ and conversely.

We recall that $P \xrightarrow{I/O} P'$ means $P \mid P_I \overset{\tau}{\Rightarrow} P''$, with $P_I = \Pi_{s\in I}\overline{s}$, $P'' \downarrow$, $O = \{s \mid P'' \xrightarrow{\overline{s}} \cdot\}$, and $P' = \lfloor P'' \rfloor$. First, note that $P \approx_L Q$ implies $P \mid P_I \approx_L Q \mid P_I$. If $(P \mid P_I) \overset{\tau}{\Rightarrow} P''$

and $P'' \downarrow$ then by $(B1)$ $Q \mid P_I \overset{\tau}{\Rightarrow} Q_1$ and $P'' \approx_L Q_1$. Moreover, by $(B2)$, $Q_1 \overset{\tau}{\Rightarrow} Q''$, $Q'' \downarrow$, $P'' \approx_L Q''$, and $P' = \lfloor P'' \rfloor \approx_L \lfloor Q'' \rfloor = Q'$. By $(B3)$, if $P'' \xrightarrow{\overline{s}} \cdot$ then $Q'' \xrightarrow{\overline{s}} \cdot$, and conversely. Thus $Q \xrightarrow{I/O} Q'$. □

Next, we recast the strong confluence result mentioned in section 3 in the following terms.

Proposition 37 *If $P \xrightarrow{\alpha_1} P_1$ and $P \xrightarrow{\alpha_2} P_2$ then either $P_1 = P_2$ or $\exists\, P_{12}$ ($P_1 \xrightarrow{\alpha_2} P_{12}$ and $P_2 \xrightarrow{\alpha_1} P_{12}$).*

We now look at some additional properties that can be derived from the strong confluence proposition 37.

Lemma 38 (1) *If $P \xrightarrow{\tau} P_1$, $P \xrightarrow{s} P_2$, and $\neg P \xrightarrow{\overline{s}} \cdot$ then $\exists\, P_{12}$ $P_1 \xrightarrow{s} P_{12}$ and $P_2 \xrightarrow{\tau} P_{12}$.*

(2) *If $P \xrightarrow{s} P'$ and $P \xrightarrow{\overline{s}} \cdot$ then $P \xrightarrow{\tau} P'$.*

(3) *If $P \xrightarrow{\tau} P_1$, $P \xrightarrow{\tau} P_2$ and $P_1 \downarrow$ then $P_1 = P_2$.*

(4) *If $P \overset{\tau}{\Rightarrow} P_1$, $P \overset{\tau}{\Rightarrow} P_2$, $P_1 \downarrow$, and $P_2 \downarrow$ then $P_1 = P_2$.*

(5) *If $P \xrightarrow{I/O_1} P_1$ and $P \xrightarrow{I/O_2} P_2$ then $P_1 = P_2$ and $O_1 = O_2$.*

Proof. We just check (5). By (4), if $P \mid P_I \overset{\tau}{\Rightarrow} P_1'$, $P_1' \downarrow$, $P \mid P_I \overset{\tau}{\Rightarrow} P_2'$, and $P_2' \downarrow$ then $P_1' = P_2'$. This forces $P_1 = \lfloor P_1' \rfloor = \lfloor P_2' \rfloor = P_2$ and $O_1 = O_2$. □

The following proposition states an interesting consequence of confluence.[4]

Proposition 39 *$P \Downarrow_L$ if and only if $P \Downarrow$.*

Proof. By definition, $P \Downarrow$ implies $P \Downarrow_L$. To show the other direction, suppose $P \Downarrow_L$ and let $P \xrightarrow{\alpha_1} P_1 \cdots \xrightarrow{\alpha_n} P_n$ be a sequence of transitions of minimal length leading to a program P_n such that $P_n \downarrow$. We build a sequence of internal transitions τ leading to a suspended program. First, we notice that the actions α_i cannot be emission actions, otherwise a shorter sequence can be found. Second, we can assume that the last action α_n is an internal transition τ. Otherwise, if $\alpha_n = s$ then either $P_{n-1} \xrightarrow{\overline{s}} \cdot$ and then $P_{n-1} \xrightarrow{\tau} P_n$ by lemma 38(1) or $\neg P_{n-1} \xrightarrow{\overline{s}} \cdot$ and then $P_{n-1} \downarrow$ contradicting the minimal length hypothesis.

Let us now look at a sequence of transitions:

$$P \xrightarrow{s} P_1 \xrightarrow{\tau} \cdots \xrightarrow{\tau} P_n \qquad n \geq 2\ . \tag{3}$$

where $\neg P \xrightarrow{\overline{s}} \cdot$ and $\neg P \downarrow$. Then we must have $P \xrightarrow{\tau} P'$ and by lemma 38(1) there is a P_1' such that $P' \xrightarrow{s} P_1'$ and $P_1 \xrightarrow{\tau} P_1'$. By the confluence properties and lemma 38(3), $P_1' \overset{\tau}{\Rightarrow} P_n$ in $n-2$ transitions τ. Thus we have the following sequence of transitions:

$$P \xrightarrow{\tau} P' \xrightarrow{s} P_1' \overset{\tau}{\Rightarrow} P_n \tag{4}$$

[4]One can conceive non-deterministic extensions of the language where the proposition fails.

The number of τ transitions that follow the s transition is $n-1$ in (3) and $n-2$ in (4). By iterating this reasoning, the input transition s is eventually removed. Moreover, the argument is extended to a sequence of transitions containing several input actions by simply removing the input actions one after the other proceeding backwards. $\Box$

In view of proposition 39, the hypothesis $P \Downarrow_L$ can be replaced by the hypothesis $P \Downarrow$ in condition $(B3)$. Now consider an alternative definition where the hypothesis $P \Downarrow_L$ is replaced by the hypothesis $P \downarrow$. We refer to this condition as $(B3)^{\downarrow}$, call the resulting notion of bisimulation $\downarrow$*-labelled bisimulation*, and denote with $\approx_L^{\downarrow}$ the related largest bisimulation.

Proposition 40 $\approx_L = \approx_L^{\downarrow}$.

This is a direct consequence of the following lemma whose proof is delayed to appendix A.9.

Lemma 41 (1) *If $P \approx_L Q$ then $P \approx_L^{\downarrow} Q$.*
(2) *The relation $\approx_L^{\downarrow}$ is reflexive and transitive.*
(3) *If $P \xrightarrow{\tau} Q$ then $P \approx_L Q$, $P \approx_L^{\downarrow} Q$, and $tr(P) = tr(Q)$.*
(4) *$\approx_L^{\downarrow}$ is a labelled bisimulation.*

We rely on this characterisation to show that bisimulation and trace equivalence collapse; an expected property of deterministic systems. To this end, we note the following properties of trace equivalence whose proof is given in appendix A.10

Lemma 42 (1) *If $tr(P) = tr(Q)$ then $tr(P \mid (\mathsf{emit}\ s)) = tr(Q \mid (\mathsf{emit}\ s))$.*
(2) *$\mathcal{R} = \{(P, Q) \mid tr(P) = tr(Q)\}$ is a labelled bisimulation.*

From proposition 36 and lemma 42(2), we derive the collapse of trace and bisimulation equivalence.

Theorem 43 *$P \approx_L Q$ if and only if $tr(P) = tr(Q)$.*

7 Conclusion

Motivated by recent developments in reactive programming, we have introduced a revised definition of the SL model including thread spawning and recursive definitions. The revised model is still confluent and therefore deterministic. We have proposed a simple static analysis that entails reactivity in the presence of recursive definitions and characterised the computational power of the model with and without signal generation. Moreover, we have identified a tail recursive core language which is built around the present operator and whose justification comes directly from the basic design principle of the SL model. The simplification of the model has been instrumental to the development of a compositional notion of program equivalence. In further investigations, we plan to extend this approach to a Synchronous Language including data values and name mobility.

Acknowledgements

The author is indebted to G. Boudol, F. Boussinot, I. Castellani, and F. Dabrowski for a number of discussions on the topic of this paper and for suggesting improvements in its presentation.

References

[1] R. Amadio, G. Boudol, F. Boussinot and I. Castellani. Reactive programming, revisited. In Proc. Workshop on *Algebraic Process Calculi: the first* 25 *years and beyond*, Bertinoro, NS-05-3 BRICS Notes Series, August 2005.

[2] R. Amadio, I. Castellani and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In *Theor. Comput. Sci.*, 195:291-324, 1998.

[3] R. Amadio, S. Dal-Zilio. Resource control for synchronous cooperative threads. In *Proc. CONCUR*, Springer LNCS 3170, 2004.

[4] R. Amadio, F. Dabrowski. Feasible reactivity for synchronous cooperative threads. In *Proc. EXPRESS*, ENTCS, 2005 (to appear).

[5] G. Berry and G. Gonthier, The Esterel synchronous programming language. *Science of computer programming*, 19(2):87–152, 1992.

[6] G. Boudol, ULM, a core programming model for global computing. In *Proc. of ESOP*, Springer LNCS 2986, 2004.

[7] F. Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

[8] F. Boussinot and R. De Simone, The SL Synchronous Language. *IEEE Trans. on Software Engineering*, 22(4):256–266, 1996.

[9] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi (extended abstract) In *Proc. ICALP*, Springer LNCS 1443, 1998.

[10] M. Hennessy and J. Rathke. Bisimulations for a calculus of broadcasting systems. In *Theor. Comput. Sci.*, 200(1-2):225-260, 1998.

[11] K. Honda and N. Yoshida. On reduction-based process semantics. In *Theor. Comput. Sci.*, 151(2): 437-486, 1995.

[12] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress, North-Holland*, 1974.

[13] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Proc. ACM Principles and Practice of Declarative Programming*, 2005.

[14] A. Matos, G. Boudol and I. Castellani. Typing non-inteference for reactive programs. RR-INRIA 5594, June 2005. Extended abstract presented at the *Foundations of Computer Security 2004* workshop.

[15] R. Milner. Communication and Concurrency. Prentice-Hall, 1989.

[16] R. Milner and D. Sangiorgi. Barbed bisimulation. In *Proc. ICALP*, Springer LNCS 623, 1992.

[17] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the USENIX Technical Conference, 1996.

[18] K.V.S. Prasad. A calculus of broadcasting systems. In *Sci. Comput. Program.*, 25(2-3): 285-327, 1995.

[19] Reactive Programming, INRIA, Mimosa Project. `http://www-sop.inria.fr/mimosa/rp`.

[20] M. Serrano, F. Boussinot, and B. Serpette. Scheme fair threads. In *Proc. ACM Principles and practice of declarative programming*, 2004.

A Proofs

A.1 Proof of proposition 1

By induction on the structure of T assuming ';' associates to the right. If $T = 0$ then clearly no decomposition is possible. If $T \neq 0$ is a redex then take $C = [\ ]$ and observe that no other context is possible. If T has the shape $\Delta; T'$ then take $C = [\ ]; T'$. If T has the shape $(\mathsf{watch}\ s\ T')$ and $T' \neq 0$ then by inductive hypothesis we have a unique decomposition $T' = C'[\Delta']$ and the only possible decomposition for T is obtained by taking $C = (\mathsf{watch}\ s\ C')$ and $\Delta = \Delta'$. Finally, if $T = (\mathsf{watch}\ s\ T'); T''$ and $T' \neq 0$ then by inductive hypothesis we have a unique decomposition $T' = C'[\Delta']$ and the only possible decomposition for T is obtained by taking $C = (\mathsf{watch}\ s\ C'); T''$ and $\Delta = \Delta'$. □

A.2 Proof of theorem 3

First we notice that the notion of reduction, suspension, and evaluation at the end of an instant can be defined up to renaming.

Proposition 44 *Suppose $(P_1, E_1) =_\alpha (P_2, E_2)$. Then the following holds.*

(1) *If $(P_1, E_1) \overset{P_1''}{\rightarrow} (P_1', E_1')$ then $(P_2, E_2) \overset{P_2''}{\rightarrow} (P_2', E_2')$ and $(P_1' \cup P_1'', E_1') =_\alpha (P_2' \cup P_2'', E_2')$.*

(2) *$(P_1, E_1) \downarrow$ if and only if $(P_2, E_2) \downarrow$.*

(3) *If $(P_1, E_1) \downarrow$ then $\lfloor P_1 \rfloor_{E_1} =_\alpha \lfloor P_2 \rfloor_{E_2}$.*

Proof. (1) By case analysis on the reduction.

(2) Suppose $T_i = C_i[\mathsf{await}\ s_i]$ for $i = 1, 2$ and σ is a renaming such that $\sigma T_1 = T_2$ and $E_1 = E_2 \circ \sigma$. Then check that $(T_1, E_1) \downarrow$ if and only if $(T_2, E_2) \downarrow$.

(3) Suppose $(T_1, E_1) =_\alpha (T_2, E_2)$ and $(T_1, E_1) \downarrow$. Proceed by induction on the structure of T_1. □

Then we check the strong confluence lemma from which determinism follows.

Lemma 45 (strong confluence) *If $(P, E) \overset{P_1''}{\rightarrow} (P_1', E_1')$, $(P, E) \overset{P_2''}{\rightarrow} (P_2', E_2')$, and $(P_1' \cup P_1'', E_1') \neq_\alpha (P_2' \cup P_2'', E_2')$ then there exist $\overline{P}_1'', \overline{P}_2'', P_{12}', E_{12}, P_{21}', E_{21}$ such that $(P_1', E_1') \overset{\overline{P}_2''}{\rightarrow} (P_{12}', E_{12})$, $(P_2', E_2') \overset{\overline{P}_1''}{\rightarrow} (P_{21}', E_{21})$, and $(P_{12}' \cup P_1'' \cup \overline{P}_2'', E_{12}) =_\alpha (P_{21}' \cup P_2'' \cup \overline{P}_1'', E_{21})$.*

Proof. It is convenient to work with a pair (P, E) such that all bound names are distinct and not in $dom(E)$. It is then possible to close the diagram directly taking $\overline{P}_2'' = P_2'', \overline{P}_1'' = P_1''$, $P_{12} = P_{21}$, $E_{12} = E_{21} = E_1 \vee E_2$, where:

$$(E_1 \vee E_2)(s) = \begin{cases} \mathit{true} & \text{if } E_1(s) = \mathit{true} \text{ or } E_2(s) = \mathit{true} \\ \mathit{false} & \text{otherwise, if } E_1(s) = \mathit{false} \text{ or } E_2(s) = \mathit{false} \\ \uparrow & \text{otherwise.} \end{cases}$$

We can then derive the initial statement by repeated application of proposition 44. □

A.3 Proof of theorem 11

First, it is useful to note the following commutation of substitution and CPS translation.

Lemma 46 $[\mathbf{s}/\mathbf{x}][\![T]\!](t,\tau) = [\![[\mathbf{s}/\mathbf{x}]T]\!](t,\tau)$, *assuming* $\{\mathbf{x}\} \cap sig(t,\tau) = \emptyset$.

Lemma 47 *Suppose* $T \ \mathcal{R}\ t$, *and* $(T,E) \xrightarrow{P} (T',E')$. *Then* $T = C[\Delta]$ *for some context* C *and redex* Δ *and exactly one of the following cases arises.*
(1) $\Delta ::= 0;T'' \mid (\mathsf{watch}\ s\ 0)$. *Then* $P=\emptyset$, $E=E'$, *and* $t = [\![T]\!](0,\epsilon) = [\![T']\!](0,\epsilon)$.

(2) $\Delta ::= \mathsf{thread}\ T''$. *Then* $P = \{\!|T''|\!\}$, $E=E'$, *and* $(t,E) = ([\![T]\!](0,\epsilon),E) \xrightarrow{\{\!|[\![T'']\!](0,\epsilon)|\!\}} ([\![T']\!](0,\epsilon),E)$.

(3) $\Delta ::= \mathsf{emit}\ s \mid \nu s\ T'' \mid A(\mathbf{s})$. *Then* $P=\emptyset$ *and* $(t,E) = ([\![T]\!](0,\epsilon),E) \xrightarrow{\emptyset} ([\![T']\!](0,\epsilon),E')$.

(4) $\Delta ::= \mathsf{await}\ s$ *and* $t = [\![T]\!](0,\epsilon)$. *Then* $P=\emptyset$, $E=E'$, *and* $(t,E) \xrightarrow{\emptyset} ([\![T']\!](0,\epsilon),E)$.

(5) $\Delta ::= \mathsf{await}\ s$ *and* $t = A$ *where* $A = [\![T]\!](0,\epsilon)$. *Then* $P=\emptyset$, $E=E'$, *and* $(t,E)(\xrightarrow{\emptyset}) \cdot (\xrightarrow{\emptyset})([\![T']\!](0,\epsilon),E)$.

Proof. We denote with π_1,π_2 the first and second projection, respectively.
(1) If $\Delta = 0;T$ then

$$\begin{array}{ll}
[\![C[0;T]]\!](0,\epsilon) & \\
= [\![0;T]\!]([\![C]\!](0,\epsilon)) & \text{(by proposition 9)} \\
= [\![T]\!]([\![C]\!](0,\epsilon)) & \text{(by CPS definition)} \\
= [\![C[T]]\!](0,\epsilon) & \text{(by proposition 9)}\ .
\end{array}$$

If $\Delta = \mathsf{watch}\ s\ 0$ let $(t,\tau) = [\![C]\!](0,\epsilon)$. Then

$$\begin{array}{ll}
[\![C[\mathsf{watch}\ s\ 0]]\!](0,\epsilon) & \\
= [\![\mathsf{watch}\ s\ 0]\!](t,\tau) & \text{(by proposition 9)} \\
= [\![0]\!](t,\tau\cdot(s,t)) & \text{(by CPS definition)} \\
= t & \text{(by CPS definition)} \\
= [\![0]\!](t,\tau) & \text{(by CPS definition)} \\
= [\![C[0]]\!](0,\epsilon) & \text{(by proposition 9)}\ .
\end{array}$$

(2) We observe:

$$\begin{array}{ll}
[\![C[\mathsf{thread}\ T'']]\!](0,\epsilon) & \\
= [\![\mathsf{thread}\ T'']\!]([\![C]\!](0,\epsilon)) & \text{(by proposition 9)} \\
= \mathsf{thread}\ [\![T'']\!](0,\epsilon).\pi_1([\![C]\!](0,\epsilon)) & \text{(by CPS definition)} \\
= \mathsf{thread}\ [\![T'']\!](0,\epsilon).[\![0]\!]([\![C]\!](0,\epsilon)) & \text{(by CPS definition)} \\
\xrightarrow{\{\!|[\![T'']\!](0,\epsilon)|\!\}} [\![C[0]]\!](0,\epsilon) & \text{(by } (t_5) \text{ and proposition 9)}
\end{array}$$

(3) The cases where $\Delta = (\mathsf{emit}\ s)$ or $\Delta = (\nu s\ T)$ are straightforward. Suppose $\Delta = A(\mathbf{s})$. Assume $(t,\tau) = [\![C]\!](0,\epsilon)$, $sig(t,\tau) = \{\mathbf{s}'\}$ and $A(\mathbf{x}) = T$ with $\{\mathbf{x}\}\cap\{\mathbf{s}'\} = \emptyset$. We consider

the equation $A^{(t,\tau)}(\mathbf{x}) = [\![T]\!](t,\tau)$ where we rely on the convention that the parameters $\mathbf{s}'$ are omitted. Now we have:

$$\begin{array}{ll}
[\![C[A(\mathbf{s})]]\!](0,\epsilon) & \\
= [\![A(\mathbf{s})]\!]([\![C]\!](0,\epsilon)) & \text{(by proposition 9)} \\
= A^{(t,\tau)}(\mathbf{s}) & \text{(by CPS definition)} \\
\xrightarrow{\emptyset} [\mathbf{s}/\mathbf{x}, \mathbf{s}'/\mathbf{s}'][\![T]\!](t,\tau) & \\
= [\![[\mathbf{s}/\mathbf{x}]T]\!](t,\tau) & \text{(by substitution lemma 46)} \\
= [\![[\mathbf{s}/\mathbf{x}]T]\!]([\![C]\!](0,\epsilon)) & \\
= [\![C[[\mathbf{s}/\mathbf{x}]T]]\!](0,\epsilon) & \text{(by proposition 9).}
\end{array}$$

(4) We observe:

$$[\![C[\mathsf{await}\ s]]\!](0,\epsilon) = [\![\mathsf{await}\ s]\!]([\![C]\!](0,\epsilon)) = \mathsf{present}\ s\ t\ b$$

where $t = \pi_1([\![C]\!](0,\epsilon)) = [\![C[0]]\!](0,\epsilon)$ and $(\mathsf{present}\ s\ t\ b, E) \xrightarrow{\emptyset} (t, E)$.

(5) First unfold $A(\mathbf{s})$ and then proceed as in case (4). □

Thus if $T\ \mathcal{R}\ t$ and T reduces then t can match the reduction and stay in the relation. The proofs of the following three lemma 48, 49, and 50 rely on similar arguments. First, we analyse the situation where t reduces.

Lemma 48 *Suppose $T\ \mathcal{R}\ t$, and $(t,E) \xrightarrow{p} (t',E')$. Then $T = C[\Delta]$ and exactly one of the following cases arises.*

(1) *$\Delta ::= \mathsf{await}\ s$ and $t = A$ where $A = [\![T]\!](0,\epsilon)$. Then $p = \emptyset$, $E = E'$ and $T\ \mathcal{R}\ t'$.*

(2) *$\Delta ::= \mathsf{await}\ s$ and $t = [\![T]\!](0,\epsilon)$. Then $p = \emptyset$, $E = E'$, and $(T,E) \xrightarrow{\emptyset} (T',E)$ with $t' = [\![T']\!](0,\epsilon)$.*

(3) *$\Delta ::= \mathsf{thread}\ T''$. Then $p = \{\![[\![T'']\!](0,\epsilon)]\!\}$, $E = E'$, and $(T,E) \xrightarrow{\{\![T'']\!\}} (T',E)$ with $t' = [\![T']\!](0,\epsilon)$.*

(4) *$\Delta ::= \mathsf{emit}\ s \mid \nu s\ T'' \mid A(\mathbf{s})$. Then $p = \emptyset$, $t = [\![T]\!](0,\epsilon)$, and $(T,E) \xrightarrow{\emptyset} (T',E')$ with $t' = [\![T']\!](0,\epsilon)$.*

(5) *$\Delta ::= 0;T'' \mid (\mathsf{watch}\ s\ 0)$. Then $p = \emptyset$, $E = E'$, $t = [\![T]\!](0,\epsilon)$ $(T,E) \xrightarrow{\emptyset} (T',E)$, $t = [\![T']\!](0,\epsilon)$, and T' is smaller than T.*

Thus if $T\ \mathcal{R}\ t$ and t reduces then T can match the reduction and stay in the relation. In the worst case, the number of reductions T has to make is proportional to its size. This is because case (5) shrinks the thread.

Lemma 49 *If $T\ \mathcal{R}\ t$ and $(T,E)\downarrow$ then exactly one of the following cases arises.*

(1) *$t = [\![T]\!](0,\epsilon)$. Then $(t,E)\downarrow$.*

(2) *$T = C[\mathsf{await}\ s]$, $t = A$, and $A = [\![T]\!](0,\epsilon)$. Then $(t,E) \xrightarrow{\emptyset} ([\![T]\!](0,\epsilon),E)$ and $([\![T]\!](0,\epsilon),E)\downarrow$.*

Thus if $T \mathcal{R} t$ and (T, E) is suspended then (t, E) is suspended too possibly up to an unfolding.

Lemma 50 *If $T \mathcal{R} t$ and $(t, E) \downarrow$ then $t = [\![T]\!](0, \epsilon)$ and exactly one of the following cases arises.*

(1) $T = 0$ *or* $T = C[\mathsf{await}\ s]$ *and* $(T, E) \downarrow$.

(2) $T = C[\Delta]$, $\Delta ::= 0; T'' \mid (\mathsf{watch}\ s\ 0)$. *Then* $(T, E) \xrightarrow{\emptyset} (C[0], E)$ *and* $t = [\![C[0]]\!](0, \epsilon)$.

Thus if $T \mathcal{R} t$ and (t, E) is suspended then (T, E) is suspended too possibly up to the reduction of redexes $0; T''$ or $(\mathsf{watch}\ s\ 0)$. Again the number of these reductions is at most proportional to the size of T. Next we look at the computation at the end of the instant.

Lemma 51 *If $T \mathcal{R} t$, $(T, E) \downarrow$, and $(t, E) \downarrow$ then $\lfloor T \rfloor_E \mathcal{R} \lfloor t \rfloor_E$.*

Proof. Exactly one of the following cases arises.

(1) $T = t = 0 = \lfloor T \rfloor_E = \lfloor t \rfloor_E$.

(2) $T = C[\mathsf{await}\ s]$, $t = [\![T]\!](0, \epsilon)$. We have to explicit the structure of t and relate it to the structure of the context. First, we notice that the context C can be written in the general form

$$C = (\mathsf{watch}\ s_1 \cdots (\mathsf{watch}\ s_n\ [\ ] U_{n+1}) U_n \cdots) U_1$$

where $U_i ::= \epsilon \mid ; T_i$ so that the presence of U_i is optional. Then we claim that t can be written as:

$$t = \mathsf{present}\ s\ t_{n+1} (\mathsf{ite}\ s_1\ t_1\ \cdots (\mathsf{ite}\ s_n\ t_n A) \cdots), \quad A = t$$

where t_i is defined inductively as follows:

$$\begin{array}{lll} t_0 & = 0, & \\ \tau_0 & = \epsilon & \\ t_{i+1} & = \left\{ \begin{array}{ll} [\![T_{i+1}]\!](t_i, \tau_i) & \text{if } U_{i+1} = ; T_{i+1} \\ t_i & \text{otherwise} \end{array} \right. & \text{for } i = 0, \ldots, n \\ \tau_{i+1} & = \tau_i \cdot (s_{i+1}, t_{i+1}) & \text{for } i = 0, \ldots, n-1 \end{array}$$

In particular, we have $[\![C]\!](0, \epsilon) = (t_{n+1}, \tau_n)$. Now two subcases can arise.

(2.1) $E(s_1) = \cdots = E(s_n) = \mathit{false}$. Then $\lfloor T \rfloor_E = T$ and $\lfloor t \rfloor_E = A$ so that thanks to the second clause in the definition of $\mathcal{R}$ we have $\lfloor T \rfloor_E \mathcal{R} \lfloor t \rfloor_E$.

(2.2) $E(s_1) = \cdots = E(s_{i-1}) = \mathit{false}$ and $E(s_i) = \mathit{true}$. Then

$$\lfloor T \rfloor_E = (\mathsf{watch}\ s_1 \cdots (\mathsf{watch}\ s_{i-1}\ 0\ U_i) U_{i-1} \cdots) U_1, \quad \text{and} \quad [\![\lfloor T \rfloor_E]\!](0, \epsilon) = t_i = \lfloor t \rfloor_E\ . \qquad \Box$$

To summarise, we have shown that the relation $\mathcal{R}$ acts as a kind of weak bisimulation with respect to reduction and suspension and that it is preserved by the computation at the end of the instant. Note that the relation $\mathcal{R}$ is immediately extended to programs in the source and target language by saying that the source program P is related to the target program p if there is a bijection i between the threads in P and those in p such that if $i(T) = t$ then $T \mathcal{R} t$.

Lemma 52 *Suppose $P \mathcal{R} p$. Then for every environment E:*
(1) If $(P,E)(\rightarrow)^*(P',E')$ and $(P',E')\downarrow$ then for some p' $(p,E)(\rightarrow)^*(p',E')$, $(p',E')\downarrow$, and $\lfloor P'\rfloor_{E'} \mathcal{R} \lfloor p'\rfloor_{E'}$.
(2) Vice versa, if $(p,E)(\rightarrow)^*(p',E')$ and $(p',E')\downarrow$ then for some P' $(P,E)(\rightarrow)^*(P',E')$, $(p',E')\downarrow$, and $\lfloor P'\rfloor_{E'} \mathcal{R} \lfloor p'\rfloor_{E'}$.

From lemma 52 we derive that if $P \mathcal{R} p$ then $tr(P) = tr(p)$ and in particular that $tr(P) = tr(\llbracket P \rrbracket)$ as required.

A.4 Proof of proposition 14

Let X be a finite set of thread identifiers. We define its *depth* as the length of the longest descending chain with respect to $\succ$. Consider an equation. $A(\mathbf{x}) = T$. The function $Call(T,\epsilon)$ implicitly associates a label $\ell \in \{\epsilon,\kappa\}$ with every occurrence of a thread identifier in T. Next consider a related equation $A^{(t,\tau)}(\mathbf{x}) = \llbracket T\rrbracket(t,\tau)$ and an occurrence of a thread identifier B in T. Two situations may arise: (1) The label associated with the occurrence of B is κ and then $A \succ B$. (2) The label associated with the occurrence of B is ϵ and then $A \succeq B$ and moreover the index (t',τ') of B in the CPS translation is either $(0,\epsilon)$ or (t,τ).

Then to compute the system of recursive equations associated with the CPS translation proceed as follows. First, compute the equations of 'index' $(0,\epsilon)$, *i.e.*, those of the shape $A^{(0,\epsilon)}(\mathbf{x}) = \llbracket T\rrbracket(0,\epsilon)$ and collect all the thread identifiers $A^{(t,\tau)}$ occurring on the right hand side with an index (t,τ) different from $(0,\epsilon)$. Continue, by computing the equations $A^{(t,\tau)} = \llbracket T\rrbracket(t,\tau)$ for the new indexes (t,τ). Then collect again the identifiers with new indexes. At each step the depth of the finite set of thread identifiers with new indexes decreases. Thus this process terminates with a finite number of recursive equations. $\Box$

A.5 Proof of theorem 19

We start by describing the simulation of simple deterministic *push down automata*. The empty stack is represented by the symbol Z. The stack alphabet has only one symbol S. A configuration of an automaton is a pair $(q, S\cdots SZ)$ composed of a state and a stack, and its possible transitions are:

$$\begin{array}{lll}
(q,w) \mapsto (q',Sw) & & \text{(increment)}\\
(q,Sw) \mapsto (q',w) & & \text{(decrement)}\\
(q,w) \mapsto \left\{\begin{array}{ll} (q',w) & w = Z \\ (q'',w) & w \neq Z \end{array}\right. & & \text{(test zero)}
\end{array}$$

We introduce as many thread identifiers as states. Each of these thread identifiers has parameters *inc*, *dec*, *zero*, *ack* which we omit. Depending on the instructions associated with the state, we introduce one of the following equations:

$$\begin{array}{lll}
q & = (\mathsf{emit}\ inc); (\mathsf{await}\ ack); \mathsf{pause}; q' & \text{(increment)}\\
q & = (\mathsf{emit}\ dec); (\mathsf{await}\ ack); \mathsf{pause}; q' & \text{(decrement)}\\
q & = (\mathsf{present}\ zero\ (\mathsf{pause}; q')\ q'') & \text{(test zero)}
\end{array}$$

Note that the control starts at most one operation per instant and that it waits for the completion of the operation before proceeding to the following one.

Next we represent the stack. This is similar to what is done, *e.g.*, in CCS [15]. We abbreviate with $\mathbf{s}$ a vector of 5 signals $dec, inc, zero, ack, abort$. A thread Z depends on such a vector for interactions on the 'left'. A thread S (or S_+, S_r, S_l) depends on a pair of vectors $\mathbf{s}, \mathbf{s}'$ for interactions on the 'left'and on the 'right', respectively.

$$\begin{array}{ll}
Z(\mathbf{s}) & = (\mathsf{watch}\ abort\ (\mathsf{emit}\ zero); \\
& \quad (\mathsf{present}\ inc \\
& \quad\quad (\mathsf{emit}\ ack); \mathsf{pause}; (\nu \mathbf{s}'\ (\mathsf{thread}\ S(\mathbf{s},\mathbf{s}'), Z(\mathbf{s}'))) \\
& \quad\quad (\mathsf{thread}\ Z(\mathbf{s}))))
\end{array}$$

$$\begin{array}{ll}
S(\mathbf{s},\mathbf{s}') & = (\mathsf{thread} \\
& \quad (\mathsf{watch}\ dec\ (\mathsf{await}\ inc); \mathsf{pause}; (\mathsf{thread}\ S_+(\mathbf{s},\mathbf{s}'))), \\
& \quad (\mathsf{watch}\ inc\ (\mathsf{await}\ dec); \mathsf{pause}; (\mathsf{thread}\ S_r(\mathbf{s},\mathbf{s}'))))
\end{array}$$

$$S_+(\mathbf{s},\mathbf{s}') = (\nu \mathbf{s}''\ (\mathsf{emit}\ ack);\ (\mathsf{thread}\ S(\mathbf{s},\mathbf{s}''), S(\mathbf{s}'',\mathbf{s}')))$$

$$\begin{array}{ll}
S_r(\mathbf{s},\mathbf{s}') & = (\mathsf{present}\ zero'\ (\mathsf{emit}\ abort'); \mathsf{pause}; (\mathsf{emit}\ ack); Z(\mathbf{s}) \\
& \quad (\mathsf{emit}\ dec'); S_l(\mathbf{s},\mathbf{s}')
\end{array}$$

$$S_l(\mathbf{s},\mathbf{s}') = (\mathsf{await}\ ack'); \mathsf{pause}; (\mathsf{emit}\ ack); S(\mathbf{s},\mathbf{s}')$$

A configuration $(q, S \cdots SZ)$ of the automaton is mapped to a program which is essentially equivalent to: $(\nu \mathbf{s}_0, \ldots, \mathbf{s}_n\ (\mathsf{thread}\ q(\mathbf{s}_0), S(\mathbf{s}_0,\mathbf{s}_1), \ldots, S(\mathbf{s}_{n-1},\mathbf{s}_n), Z(\mathbf{s}_n)))$. It is not difficult to check that the program can simulate the transitions of the automata (and this is all we need to check since the program is deterministic!). The more complex dynamics, is introduced by the decrement. Roughly, the decrement of a stack represented by the threads S, S, S, Z goes through the following transformations:

$$S,S,S,Z \to S_r,S,S,Z \to S_l,S_r,S,Z \to S_l,S_l,S_r,Z \to S_l,S_l,Z \to S_l,S,Z \to S,S,Z$$

There is a wave from left to right that transforms S into S_l, when the wave meets Z, it aborts Z, transforms the rightmost S into Z, and produces a wave from right to left that turns S_l into S again. The simulating program can be put in tail recursive form via the CPS translation. In particular, note that all recursive calls in the scope of a watch are under a thread statement that has the effect of resetting the evaluation context. Finally, we remark that the simulation of deterministic push down automata can be easily generalised to deterministic two counters machines by simply letting the control operate on two distinct stacks. □

A.6 Proof of proposition 20

(1) By induction on the proof of $P \xrightarrow{\overline{s}} P'$.

(2) If $P \xrightarrow{\overline{s}} \cdot$ then P has the shape $D[\mathsf{emit}\ s]$ for a suitable context D built out of restrictions and parallel compositions. It is easily checked that after a transition the emission $\mathsf{emit}\ s$ is still observable.

(3) By induction on the proof of $P \xrightarrow{s} P'$. □

A.7 Proof of lemma 31

Most properties follow by routine verifications. We just highlight some points.

(1) Recalling that $P \equiv_L Q$ and $P \downarrow$ implies $Q \downarrow$.

(2) Condition $(S1)$ entails conditions $(B1)$, $(B3)$, and $(L2)$, while condition $(S2)$ (with $(S1)$) entails conditions $(B2)$ and $(L1)$.

(3) Introduce a notion of normalised program where parallel composition associates to the left, all restrictions are carried at top level, and 0 programs are removed. Then define a relation R where two programs are related if their normalised forms are identical up to bijective permutations of the restricted names and the parallel components. A pair of programs equated by the laws under consideration is in R. Show that R is a strong labelled bisimulation.

(4) Show that $\{(P \mid S, Q \mid S) \mid P \equiv_L Q\}$ is a strong labelled bisimulation where S is defined as in the statement.

(5) Direct diagram chasing.

(6) We reason up to $\equiv_L$.

(7) We show $\{(P_1 \mid Q, P_2 \mid Q) \mid P_1 \equiv_L P_2\}$ is a strong labelled bisimulation up to $\equiv_L$. Let us focus on condition $(S2)$. Let $X = \{s' \mid (P_1 \mid P_2) \xrightarrow{\overline{s'}} \cdot\}$ and let S' be the parallel composition of the emissions $(\mathsf{emit}\ s)$ where $s \in X$. Suppose $(P_1 \mid Q \mid S) \downarrow$. Then we note that $P_1 \mid Q \mid S \equiv_L (P_1 \mid S' \mid S) \mid (Q \mid S' \mid S)$ and $\lfloor P_1 \mid Q \mid S \rfloor \equiv_L \lfloor P_1 \mid S' \mid S \rfloor \mid \lfloor Q \mid S' \mid S \rfloor$. A similar remark applies to $P_2 \mid Q$. Then we can conclude by reasoning up to $\equiv_L$. □

A.8 Proof of lemma 35

(1) We show that the relation $R = \approx_L \cup \{(\ P \mid (\mathsf{emit}\ s), Q \mid (\mathsf{emit}\ s)\ ) \mid P \approx_L Q\}$ is a labelled bisimulation up to $\equiv_L$. We assume $P \approx_L Q$ and we analyse the conditions $(B1-3)$ and $(L1-2)$.

$(B1)$ Suppose $P \mid (\mathsf{emit}\ s) \xrightarrow{\tau} P' \mid (\mathsf{emit}\ s)$. If the action τ is performed by P then the hypothesis and condition (B1) allow to conclude. Otherwise, suppose $P \xrightarrow{s} P'$. Then we apply the hypothesis and condition $(L2)$. Two cases may arise: (1) If $Q \overset{s}{\Rightarrow} Q'$ and $P' \approx_L Q'$ then the conclusion is immediate. (2) If $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \approx_L Q' \mid (\mathsf{emit}\ s)$ then we note that $Q' \mid (\mathsf{emit}\ s) \equiv_L (Q' \mid (\mathsf{emit}\ s)) \mid (\mathsf{emit}\ s)$ and we close the diagram up to $\equiv_L$.

$(B3)$ Suppose $P \mid (\mathsf{emit}\ s) \xrightarrow{\overline{s'}} \cdot$ and $P \mid (\mathsf{emit}\ s) \Downarrow_L$. If $s = s'$ then $Q \mid (\mathsf{emit}\ s) \xrightarrow{\overline{s'}} \cdot$ and we are done. Otherwise, it must be that $P \xrightarrow{\overline{s'}} \cdot$. Moreover, $P \Downarrow_L$. Then $P \approx_L Q$ and condition $(B3)$ imply that $Q \overset{\tau}{\Rightarrow} Q' \xrightarrow{\overline{s'}} \cdot$, and $P \approx_L Q'$. Hence $Q \mid (\mathsf{emit}\ s) \overset{\tau}{\Rightarrow} Q' \mid (\mathsf{emit}\ s) \xrightarrow{\overline{s'}} \cdot$ and we can conclude.

$(L1)$ Suppose $S = (\mathsf{emit}\ s_1) \mid \cdots \mid (\mathsf{emit}\ s_n)$. Define $S' = (\mathsf{emit}\ s) \mid S$. Then $P \approx_L Q$ and condition $(L1)$ applied to S' allows to conclude.

$(L2)$ Suppose $P \mid (\mathsf{emit}\ s) \xrightarrow{s'} P' \mid (\mathsf{emit}\ s)$. Necessarily $P \xrightarrow{s'} P'$. Given $P \approx_L Q$ and condition $(L2)$ two cases may arise: (1) $Q \overset{s'}{\Rightarrow} Q'$ and $P' \approx_L Q'$. Then the conclusion is immediate. (2) $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \approx_L Q' \mid (\mathsf{emit}\ s')$. Then $Q \mid (\mathsf{emit}\ s) \overset{\tau}{\Rightarrow} Q' \mid (\mathsf{emit}\ s)$ and we observe that $(Q' \mid (\mathsf{emit}\ s)) \mid (\mathsf{emit}\ s') \equiv_L (Q' \mid (\mathsf{emit}\ s')) \mid (\mathsf{emit}\ s)$ thus closing the diagram up to $\equiv_L$.

(2) It is easily checked that the identity relation is a labelled bisimulation. Reflexivity follows. As for transitivity, we check that the relation $\approx_L \circ \approx_L$ is a labelled bisimulation up to $\equiv_L$.

$(B1-3, L1)$ These cases are direct. For $(B3)$, recall proposition 29(2).

$(L2)$ Suppose $P_1 \approx_L P_2 \approx_L P_3$ and $P_1 \xrightarrow{s} P_1'$. Two interesting cases arise when either P_2 or P_3 match an input action with an internal transition. (1) Suppose first $P_2 \overset{\tau}{\Rightarrow} P_2'$ and $P_1 \approx_L P_2' \mid (\mathsf{emit}\ s)$. By $P_2 \approx_L P_3$ and repeated application of $(B1)$ we derive that $P_3 \overset{\tau}{\Rightarrow} P_3'$ and $P_2' \approx_L P_3'$. By property (1) the latter implies that $P_2' \mid (\mathsf{emit}\ s) \approx_L P_3' \mid (\mathsf{emit}\ s)$ and we combine with $P_1 \approx_L P_2' \mid (\mathsf{emit}\ s)$ to conclude. (2) Next suppose $P_2 \overset{\tau}{\Rightarrow} P_2^1 \xrightarrow{s} P_2^2 \overset{\tau}{\Rightarrow} P_2'$ and $P_1 \approx_L P_2'$. Suppose that P_3 matches these transitions as follows: $P_3 \overset{\tau}{\Rightarrow} P_3^1 \overset{\tau}{\Rightarrow} P_3^2$, $P_2^2 \approx_L P_3^2 \mid (\mathsf{emit}\ s)$, and moreover $P_3^2 \mid (\mathsf{emit}\ s) \overset{\tau}{\Rightarrow} P_3' \mid (\mathsf{emit}\ s)$ with $P_2' \approx_L P_3' \mid (\mathsf{emit}\ s)$. Two subcases may arise: (i) $P_3^2 \overset{\tau}{\Rightarrow} P_3'$. Then we have $P_3 \overset{\tau}{\Rightarrow} P_3'$, $P_2' \approx_L P_3' \mid (\mathsf{emit}\ s)$ and we can conclude. (ii) $P_3^2 \overset{s}{\Rightarrow} P_3'$. Then we have $P_3 \overset{s}{\Rightarrow} P_3'$ and $P_2' \approx_L P_3' \mid (\mathsf{emit}\ s) \equiv_L P_3'$.

(3) We show that $\{(\nu s\ P, \nu s\ Q) \mid P \approx_L Q\}$ is a labelled bisimulation up to $\equiv_L$.

$(B1)$ If $\nu s\ P \xrightarrow{\tau} P''$ then $P'' = \nu s P'$ and $P \xrightarrow{\tau} P'$. From $P \approx_L Q$ and $(B1)$ we derive $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \approx_L Q'$. Then $\nu s\ Q \overset{\tau}{\Rightarrow} \nu s\ Q'$ and we conclude.

$(B3)$ If $\nu s\ P \xrightarrow{\overline{s'}} \cdot$ $(s \neq s')$ then $P \xrightarrow{\overline{s'}} \cdot$. From $P \approx_L Q$ and $(B3)$ we derive $Q \overset{\tau}{\Rightarrow} Q'$, $Q' \xrightarrow{\overline{s'}} \cdot$, and $P \approx_L Q'$. To conclude, note that $\nu s\ Q \overset{\tau}{\Rightarrow} \nu s\ Q'$ and $\nu s\ Q' \xrightarrow{\overline{s'}} \cdot$.

$(L1)$ Let $S = (\mathsf{emit}\ s_1) \mid \cdots \mid (\mathsf{emit}\ s_n)$ with $s \neq s_i$ for $i = 1, \ldots, n$. If $((\nu s\ P) \mid S) \downarrow$ then $(P \mid S) \downarrow$. From $P \approx_L Q$ and $(L1)$ we derive $(Q \mid S) \overset{\tau}{\Rightarrow} (Q' \mid S)$, $(Q' \mid S) \downarrow$, $(P \mid S) \approx_L (Q' \mid S)$, and $\lfloor P \mid S \rfloor \approx_L \lfloor Q' \mid S \rfloor$. This implies that $((\nu s\ Q) \mid S) \overset{\tau}{\Rightarrow} ((\nu s\ Q') \mid S)$ and $((\nu s\ Q') \mid S) \downarrow$. We observe that $((\nu s\ P) \mid S) \equiv_L \nu s\ (P \mid S)$, $((\nu s\ Q') \mid S) \equiv_L \nu s\ (Q' \mid S)$, $\lfloor (\nu s\ P) \mid S \rfloor \equiv_L \nu s\ \lfloor P \mid S \rfloor$, and $\lfloor (\nu s\ Q') \mid S \rfloor \equiv_L \nu s\ \lfloor Q' \mid S \rfloor$. Then we can close the diagram up to $\equiv_L$.

$(L2)$ Suppose $\nu s\ P \xrightarrow{s'} P''$. Then $s \neq s'$ and $P'' = \nu s\ P'$ with $P \xrightarrow{s'} P'$. From $P \approx_L Q$ and $(L2)$ two cases may arise. (1) If $Q \overset{s'}{\Rightarrow} Q'$ and $P' \approx_L Q'$ then $\nu s\ Q \overset{s'}{\Rightarrow} \nu s\ Q'$ and we are done. (2) If $Q \overset{\tau}{\Rightarrow} Q'$ and $P' \approx_L Q' \mid (\mathsf{emit}\ s')$ then $\nu s\ Q \overset{\tau}{\Rightarrow} \nu s\ Q'$ and we note that $\nu s\ Q' \mid (\mathsf{emit}\ s') \equiv_L \nu s\ (Q' \mid (\mathsf{emit}\ s'))$ thus closing the diagram up to $\equiv_L$.

(4) We show that $R = \{(P_1 \mid Q, P_2 \mid Q) \mid P_1 \approx_L P_2\} \cup \approx_L$ is a labelled bisimulation up to $\equiv_L$.

$(B1)$ Suppose $(P_1 \mid Q) \xrightarrow{\tau} P'$.

$(B1)[1]$ If the τ transition is due to P_1 or Q then the corresponding P_2 or Q matches the transition and we are done.

$(B1)[2]$ Otherwise, suppose $P_1 \xrightarrow{s} P_1'$ and $Q \xrightarrow{\overline{s}} Q$.

$(B1)[2.1]$ If $P_2 \overset{s}{\Rightarrow} P_2'$ and $P_1' \approx_L P_2'$ then $(P_2 \mid Q) \overset{\tau}{\Rightarrow} (P_2' \mid Q)$ and we are done.

$(B1)[2.2]$ If $P_2 \overset{\tau}{\Rightarrow} P_2'$ and $P_1' \approx_L (P_2' \mid (\mathsf{emit}\ s))$ then $(P_2 \mid Q) \overset{\tau}{\Rightarrow} (P_2' \mid Q)$ and $((P_2' \mid Q) \mid (\mathsf{emit}\ s)) \equiv_L ((P_2' \mid (\mathsf{emit}\ s)) \mid Q)$ so that we close the diagram up to $\equiv_L$.

$(B1)[3]$ Otherwise, suppose $P_1 \xrightarrow{\overline{s}} P_1$ and $Q \xrightarrow{s} Q'$.

$(B1)[3.1]$ If $\neg P_1 \Downarrow_L$ then by lemma 29, $\neg(P_1 \mid Q) \Downarrow_L$, $\neg(P_1 \mid Q') \Downarrow_L$, $\neg P_2 \Downarrow_L$, $\neg(P_2 \mid Q) \Downarrow_L$. Therefore $(P_1 \mid Q') \approx_L (P_2 \mid Q)$.

$(B1)[3.2]$ If $P_1 \Downarrow_L$ then $P_2 \overset{\overline{s}}{\Rightarrow} P_2'$ and $P_1 \approx_L P_2'$. Hence $(P_2 \mid Q) \overset{\tau}{\Rightarrow} (P_2' \mid Q')$ and $(P_1 \mid Q')\ \mathcal{R}\ (P_2' \mid Q')$.

$(B3)$ Suppose $(P_1 \mid Q) \Downarrow_L$.

$(B3)[1]$ Suppose $P_1 \xrightarrow{\overline{s}} \cdot$. Then $P_1 \Downarrow_L$ and by $(B3)$ $P_2 \overset{\tau}{\Rightarrow} P_2' \xrightarrow{\overline{s}} \cdot$ and $P_1 \approx_L P_2'$. Thus $(P_2 \mid Q) \overset{\tau}{\Rightarrow} (P_2' \mid Q) \xrightarrow{\overline{s}} \cdot$ and we can conclude.

$(B3)[2]$ Suppose $Q \xrightarrow{\overline{s}}$. Then $(P_2 \mid Q) \xrightarrow{\overline{s}}$ and we are done.

$(L1)$ Suppose $(P_1 \mid Q \mid S) \downarrow$. Then $(P_1 \mid S) \downarrow$ and from $P_1 \approx_L P_2$ we derive $(P_2 \mid S) \overset{\tau}{\Rightarrow} (P_2' \mid S) \downarrow$ and $(P_1 \mid S) \approx_L (P_2' \mid S)$. In particular, $\{s \mid P_1 \mid S \xrightarrow{\overline{s}} \cdot\} = \{s \mid P_2' \mid S \xrightarrow{\overline{s}} \cdot\}$. We can also derive that $(P_2 \mid Q \mid S) \overset{\tau}{\Rightarrow} (P_2' \mid Q \mid S)$, however $(P_2' \mid Q \mid S) \downarrow$ may fail because of a synchronisation of P_2' and Q on some signal which is not already in S. Then we consider S' as the parallel composition of emissions $(\mathsf{emit}\ s)$ where $(P_1 \mid Q) \xrightarrow{\overline{s}} \cdot$. By lemma 31, we derive that:

$$\begin{array}{lll} (i) & (P_1 \mid Q \mid S) & \equiv_L (P_1 \mid S \mid S') \mid (Q \mid S \mid S') \quad \text{and} \\ (ii) & (P_2' \mid Q \mid S) & \equiv_L (P_2' \mid S \mid S') \mid (Q \mid S \mid S')\ . \end{array}$$

We also observe that $(P_1 \mid S \mid S') \downarrow$. Together with $(P_1 \mid S) \approx_L (P_2' \mid S)$ this implies by $(L1)$ $(P_2' \mid S \mid S') \overset{\tau}{\Rightarrow} (P_2'' \mid S \mid S') \downarrow$, $(P_1 \mid S \mid S') \approx_L (P_2'' \mid S \mid S')$, and $\lfloor P_1 \mid S \mid S' \rfloor \approx_L \lfloor P_2'' \mid S \mid S' \rfloor$. Now it must be that $((P_2'' \mid S \mid S') \mid (Q \mid S \mid S')) \downarrow$ because the left component already emits all the signals that could be emitted by the right one (and vice versa). By conditions $(S1-2)$ and *(ii)* we have that $(P_2' \mid Q \mid S) \overset{\tau}{\Rightarrow} (P_2''' \mid Q \mid S) \downarrow$ and $(P_2''' \mid Q \mid S) \equiv_L (P_2'' \mid S \mid S') \mid (Q \mid S \mid S')$. To summarise, we have shown that $(P_2 \mid Q \mid S) \overset{\tau}{\Rightarrow} (P_2''' \mid Q \mid S) \downarrow$,

$$\begin{array}{l} (P_1 \mid Q \mid S) \equiv_L (P_1 \mid S \mid S') \mid (Q \mid S \mid S')\ \mathcal{R}\ (P_2'' \mid S \mid S') \mid (Q \mid S \mid S') \equiv_L (P_2''' \mid Q \mid S),\ \text{and} \\ \lfloor P_1 \mid Q \mid S \rfloor \equiv_L \lfloor P_1 \mid S \mid S' \mid Q \mid S \mid S' \rfloor\ \mathcal{R}\ \lfloor P_2'' \mid S \mid S' \mid Q \mid S \mid S' \rfloor \equiv_L \lfloor P_2''' \mid Q \mid S \rfloor \end{array}$$

as required by the notion of labelled bisimulation up to $\equiv_L$.

$(L2)$ Suppose $P_1 \mid Q \xrightarrow{s} P'_1 \mid Q$.

$(L2)[1]$ Suppose $P_1 \xrightarrow{s} P'_1$.

$(L2)[1.1]$ If $P_2 \overset{s}{\Rightarrow} P'_2$ and $P'_1 \approx_L P'_2$ we are done.

$(L2)[1.2]$ If $P_1 \overset{\tau}{\Rightarrow} P'_2$ and $P'_1 \approx_L P'_2 \mid (\mathsf{emit}\ s)$ then $P_2 \mid Q \overset{\tau}{\Rightarrow} P'_2 \mid Q$ and we note that $(P'_2 \mid Q) \mid (\mathsf{emit}\ s) \equiv_L (P'_2 \mid (\mathsf{emit}\ s)) \mid Q$.

$(L2)[2]$ Suppose $Q \xrightarrow{s} Q'$. Then $(P_2 \mid Q) \xrightarrow{s} (P_2 \mid Q')$ and we are done.

(5) Let $Q = \mathsf{present}\ s\ P\ B$ and $Q' = \mathsf{present}\ s\ P'\ B'$.

$(B1)$ Note that $\neg(Q \xrightarrow{\tau} \cdot)$.

$(B3)$ Note that $\neg(Q \xrightarrow{\bar{s}} \cdot)$.

$(L1)$ Suppose $S = \mathsf{emit}\ s_1 \mid \cdots \mid \mathsf{emit}\ s_n$ and that $(Q \mid S) \downarrow$. Then $s_i \neq s$ for $i = 1, \ldots, n$ and $\lfloor Q \mid S \rfloor = \langle\!| B |\!\rangle_{\{s_1,\ldots,s_n\}}$. Note that $(Q' \mid S) \downarrow$ too, and from the hypothesis $B \approx_L B'$ we derive $\lfloor Q \mid S \rfloor \approx_L \lfloor Q' \mid S \rfloor = \langle\!| B |\!\rangle_{\{s_1,\ldots,s_n\}}$.

$(L2)$ The transition $\mathsf{present}\ s\ P\ B \xrightarrow{s} P \mid (\mathsf{emit}\ s)$ is matched by $\mathsf{present}\ s\ P'\ B' \xrightarrow{s} P' \mid (\mathsf{emit}\ s)$. By hypothesis, $P \approx_L P'$ and by (1), we derive $P \mid (\mathsf{emit}\ s) \approx_L P' \mid (\mathsf{emit}\ s)$. □

A.9 Proof lemma 41

(1) Condition $(B3)^{\downarrow}$ is weaker than condition $(B3)$. Therefore, $P \approx_L Q$ implies $P \approx^{\downarrow}_L Q$.

(2) Reflexivity is obvious. For transitivity, as usual, we have to check that $\approx^{\downarrow}_L \circ \approx^{\downarrow}_L$ is a $\downarrow$-labelled bisimulation. We focus on the new condition $(B3)^{\downarrow}$. Suppose $P_1 \approx^{\downarrow}_L P_2 \approx^{\downarrow}_L P_3$, $P_1 \downarrow$, and $P_1 \xrightarrow{\bar{s}} \cdot$. By $(B3)^{\downarrow}$, $P_2 \overset{\tau}{\Rightarrow} P'_2$ and $P'_2 \xrightarrow{\bar{s}} \cdot$. By $(B2)$, $P_2 \overset{\tau}{\Rightarrow} P''_2$, $P''_2 \downarrow$, and $P_1 \approx^{\downarrow}_L P''_2$. By confluence, $P'_2 \overset{\tau}{\Rightarrow} P''_2$ and $P''_2 \xrightarrow{\bar{s}} \cdot$. By $(B1)$, $P_3 \overset{\tau}{\Rightarrow} P'_3$ and $P''_2 \approx^{\downarrow}_L P'_3$. By $(B3)^{\downarrow}$, $P'_3 \overset{\tau}{\Rightarrow} P''_3$, $P''_2 \approx^{\downarrow}_L P''_3$, and $P''_3 \xrightarrow{\bar{s}} \cdot$. Thus we have that $P_3 \overset{\tau}{\Rightarrow} P''_3$, $P''_3 \xrightarrow{\bar{s}} \cdot$, and $P_1 \approx^{\downarrow}_L P''_2 \approx^{\downarrow}_L P''_3$ as required by condition $(B3)^{\downarrow}$.

(3) We check that:

$$\mathcal{R} = Id \cup \{(P, Q) \mid P \xrightarrow{\tau} Q \text{ or } Q \xrightarrow{\tau} P\}$$

is a labelled bisimulation up to $\equiv_L$, where Id is the identity relation. Thus $P \xrightarrow{\tau} Q$ implies $P \approx_L Q$. By (1), $P \approx^{\downarrow}_L Q$ and by proposition 36, $tr(P) = tr(Q)$.

$(B1)$ Suppose $P \xrightarrow{\tau} P_1$. If $P \xrightarrow{\tau} Q$ then by confluence, either $P_1 = Q$ or $\exists\, P_{12}\ \ P_1 \xrightarrow{\tau} P_{12}$ and $Q \xrightarrow{\tau} P_{12}$. In the first case, $Q \overset{\tau}{\Rightarrow} Q$ and $(P_1, Q) \in \mathcal{R}$. In the second case, $Q \overset{\tau}{\Rightarrow} P_{12}$ and $(P_1, P_{12}) \in \mathcal{R}$. On the other hand, if $Q \xrightarrow{\tau} P$ then $Q \overset{\tau}{\Rightarrow} P_1$.

$(B3)$ Suppose $P \Downarrow_L$ and $P \xrightarrow{\bar{s}} \cdot$. If $P \xrightarrow{\tau} Q$ then $Q \xrightarrow{\bar{s}} \cdot$ and $Q \overset{\tau}{\Rightarrow} Q$. On the other hand, if $Q \xrightarrow{\tau} P$ then $Q \overset{\tau}{\Rightarrow} P$.

$(L1)$ If $P \xrightarrow{\tau} Q$ then $P \mid S \downarrow$ is impossible. On the other hand, if $Q \xrightarrow{\tau} P$ and $P \mid S \downarrow$ then $Q \mid S \overset{\tau}{\Rightarrow} P \mid S$.

$(L2)$ Suppose $P \xrightarrow{s} P_1$. If $P \xrightarrow{\tau} Q$ then either $P_1 = Q$ or $\exists\, P_{12}\;\; P_1 \xrightarrow{\tau} P_{12}$ and $Q \xrightarrow{s} P_{12}$. In the first case, we have $Q \overset{\tau}{\Rightarrow} Q$ and $P_1 \;\mathcal{R}\; Q \equiv_L Q \mid (\mathsf{emit}\ s)$. In the second case, $Q \overset{s}{\Rightarrow} P_{12}$ and $(P_1, P_{12}) \in \mathcal{R}$. On the other hand, if $Q \xrightarrow{\tau} P$ then $Q \overset{s}{\Rightarrow} P_1$.

(4) Obviously, the critical condition to check is $(B3)$. By proposition 39 we can use the predicate $\Downarrow$ rather than the predicate $\Downarrow_L$. So suppose $P_1 \approx_L Q_1$, $P_1 \xrightarrow{\overline{s}} \cdot$, $P_1 \overset{\tau}{\Rightarrow} P_2$, and $P_2 \downarrow$. By $(B1)$, $Q_1 \overset{\tau}{\Rightarrow} Q_2$ and $P_2 \approx_L^{\downarrow} Q_2$. By $(B3)^{\downarrow}$, $Q_2 \overset{\tau}{\Rightarrow} Q_3$, $Q_3 \xrightarrow{\overline{s}} \cdot$, and $P_2 \approx_L^{\downarrow} Q_3$. By (3), $P_1 \approx_L^{\downarrow} P_2$. By transitivity of $\approx_L^{\downarrow}$, $P_1 \approx_L^{\downarrow} Q_3$. $\square$

A.10 Proof of lemma 42

(1) This follows from the remark that $P \mid (\mathsf{emit}\ s) \xrightarrow{I/O} P'$ if and only if $P \xrightarrow{I \cup \{s\}/O} P'$.

(2) We check the 5 conditions.

$(B1)$ If $P \xrightarrow{\tau} P'$ then $tr(P) = tr(P')$, by lemma 41(3). Thus $Q \overset{\tau}{\Rightarrow} Q$ and $(P', Q) \in \mathcal{R}$.

$(B3)$ In view of proposition 40, it is enough to check condition $(B3)^{\downarrow}$. If $P \downarrow$ and $P \xrightarrow{\overline{s}} \cdot$ then $P \xrightarrow{\emptyset/O} \lfloor P \rfloor$ and $s \in O$. Thus $Q \xrightarrow{\emptyset/O} Q'$. In particular, $Q \overset{\tau}{\Rightarrow} Q''$, $Q'' \xrightarrow{\overline{s}}$. By lemma 41(3), $tr(Q) = tr(Q'')$. Thus $(P, Q'') \in \mathcal{R}$.

$(L1)$ If $P \mid S \downarrow$ then $P \xrightarrow{I/O} P'$ where $I = \{s \mid S \xrightarrow{\overline{s}} \cdot\}$, $P \mid S \overset{\tau}{\Rightarrow} P''$, $P'' \downarrow$, $O = \{s \mid P'' \xrightarrow{\overline{s}} \cdot\}$, and $P' = \lfloor P'' \rfloor$. By (1), $tr(P \mid S) = tr(Q \mid S)$. Thus $Q \xrightarrow{I/O} Q'$ where $Q \mid S \overset{\tau}{\Rightarrow} Q''$, $Q'' \downarrow$, and $Q' = \lfloor Q'' \rfloor$. Now $(P'', Q''), (P', Q') \in \mathcal{R}$ since by lemma 41(3) $tr(P'') = tr(P \mid S) = tr(Q \mid S) = tr(Q'')$.

$(L2)$ If $P \xrightarrow{s} P'$ then $(P \mid (\mathsf{emit}\ s)) \xrightarrow{\tau} (P' \mid (\mathsf{emit}\ s))$ and by lemma 41(3) $tr(P \mid \overline{s}) = tr(P' \mid (\mathsf{emit}\ s))$. Moreover, $P' \approx_L (P' \mid (\mathsf{emit}\ s))$ thus by proposition 36, $tr(P') = tr(P' \mid (\mathsf{emit}\ s))$. By (1), $tr(P \mid (\mathsf{emit}\ s)) = tr(Q \mid (\mathsf{emit}\ s))$. We can conclude by considering that $Q \overset{\tau}{\Rightarrow} Q$ and $(P', Q \mid (\mathsf{emit}\ s)) \in \mathcal{R}$ since $tr(P') = tr(P' \mid (\mathsf{emit}\ s)) = tr(P \mid (\mathsf{emit}\ s)) = tr(Q \mid (\mathsf{emit}\ s))$. $\square$