Communication Topology Analysis for Concurrent Programs

Matthieu Martel and Marc Gengler

Laboratoire d'Informatique de Marseille (LIM)
Parc Scientifique et Technologique de Luminy
163, avenue de Luminy - Case 901 F
13288 Marseille Cedex 9, France
E-mail: [Matthieu.Martel,Marc.Gengler]@esil.univ-mrs.fr

Abstract. In this article, we address the problem of statically determining an approximation of the communication topology of concurrent programs. These programs may contain dynamic process and channel creations and may communicate channel names as well as functions, possibly containing other communications.
We introduce a control flow analysis which builds finite state automata to improve its precision. The method is twofold. First, we build an automaton for each process in the concurrent system yielding an approximation of how the synchronizations realized by the sequential components are ordered. Second, we extract the communication topology from a reduced product automaton, which size is polynomial in the size of the original program. This analysis was implemented and we apply it to the verification of a circuit allocation mechanism.

1 Introduction

Static analysis is a widely used technique to establish properties satisfied by programs, independently of a given execution context. Most common applications include compile-time optimizations and program verification. However, this scope is extended by concurrency which introduces new problems as well as new applications. Recently, much research has been done in this area, including Amtoft et al. [2], Bodei et al. [4, 5], Jagannathan [10], Kobayashi et al. [11], Marinescu and Goldberg [12], Nielson and Nielson [14–16, 18].

In this article, we present a static analysis which computes a fine approximation of the communication topology of Concurrent ML programs [22] and, more generally, which analyzes the whole synchronizations of concurrent programs. By determining, for each reception, the set of possibly matching emissions, we find an approximation of the possibly received values.

Our analysis is a control flow analysis (CFA). Concerning sequential functional languages, the aim of a CFA is to detect which functions an expression can evaluate to [17, 19, 23, 24]. For higher order concurrent functional languages, such as Concurrent ML [22], this task is more complicated. Channel names may

be dynamically created and communicated anywhere in the program, new processes can be dynamically created and functions are first-class values which may be communicated. Hence, a piece of code in the source program (possibly containing communications) may be executed by any process. Such analyses have been proposed by Bodei et al. [4], Colby [6], Solberg et al. [25] and Mercouroff [13]. However, the precision of the CFA is closely related to the approximations made on the topology of the communications. Hence, we address the problem of minimizing for each emission, the set of possible receptors. Eliminating some impossible communications improves the annotations at reception points and, consequently, on the sequential parts of the program using the received values.

From a technical point of view, we proceed as follows. First, we order the synchronization primitives of the sequential processes in the system. This is done by building a finite automaton $\widehat{\mathcal{A}}_p$ for each process p. A labeled path denotes one possible sequence of synchronizations in p. Second, we approximate how the different processes may interact altogether by building the product of the $\widehat{\mathcal{A}}_p$'s. This product automaton possibly has size exponential in the size of the program. Hence, we introduce a reduced product automaton size is polynomial and which conservatively approximates the product automaton.

Applications of CFA for concurrent languages are twofold. First, mixing a CFA with another static analysis usually improves the precision of this latter. For instance, partial evaluation [7] of concurrent languages has been discussed by Marinescu and Goldberg [12] and by Gengler and Martel [8, 9]. Marinescu and Goldberg [12] also introduce a binding time analysis (BTA) which collects informations used by the partial evaluator. However, this BTA makes rough approximations due to the lack of informations about the topology of communications. Using a CFA would improve the performances of the partial evaluator. Second, as outlined by Bodei et al. [4], another application of CFA for concurrent languages are security and program verification. For a concurrent program, a CFA allows one to statically determine whether two given processes P_1 and P_2 may communicate together at one stage of their execution, or to statically approximate the set of channels which are shared by P_1 and P_2 during their execution. For instance, these results enable one to check access rights properties.

Section 2 gives an informal overview of the techniques developed in this article. Section 3 briefly introduces the subset of Concurrent ML we use. We introduce the analysis in Sections 4 and 5 First, in Section 4 we define an analysis for the individual processes and second, in Section 5, we introduce two ways to analyze a pool of processes, respectively based on product and reduced product automata. This analysis was implemented and results are discussed in Section 6 for a virtual circuit allocation mechanism, similar to the one used in ATM.

2 General Description

To compute precise annotations, a CFA has to use a fine approximation of the topology of the communications realized by a program. In this Section, we illustrate how our analysis works using the concurrent program of Figure 1 a).

In this system, Processes p_1 and p_3 respectively realize two and one emissions, and p_2 realizes three receptions. We assume that all the communications are made on the same channel. Communications are synchronous, i.e. an emission or a reception blocks the process until the communication occurs. In the remainder of this article, we use the following vocabulary. A *communication* is the synchronous interaction between an *emission point* and a *reception point*. A *synchronization* is either a communication or a new process creation (a `fork`). A *synchronization point* is either a communication point or a `fork` point.

A first way to conservatively annotate p_2 is to consider that any reception may receive its value from any emission on the same channel [25]. Let us call s_1 and s_2 the emissions of p_1, s_3 the emission of p_3, and r_1, r_2 and r_3 the receptions of p_2. Let $\widehat{\mathcal{C}}(s_i)$ (resp. $\widehat{\mathcal{C}}(r_j)$) be the abstract value sent (resp. received) at s_i (resp. r_j), $1 \leq i, j \leq 3$. Such an analysis assigns to $\widehat{\mathcal{C}}(r_1)$, $\widehat{\mathcal{C}}(r_2)$ and $\widehat{\mathcal{C}}(r_3)$ the values

$$\widehat{\mathcal{C}}(r_j) \;=\; \bigcup_{i \in \{1,2,3\}} \widehat{\mathcal{C}}(s_i) \quad j \in \{1,2,3\} \tag{1}$$

As sketched in Figure 1 b), Equation (1) describes a correct approximation of the communication scheme of Figure 1 a). However, it is a trivial matter of fact that s_1 cannot communicate with r_3 nor s_2 with r_1.

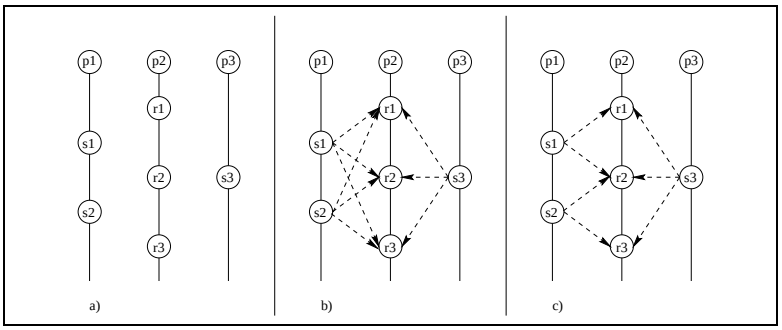


Fig. 1. a) A system made of communicating processes. b) Approximation of the communication topology computed by classical methods. c) Approximation of the topology computed by our CFA.

In order to assign a fine abstract value to receptions, a CFA has to approximate closely the communication topology of programs. This problem is twofold and we address it as follows.

– Obviously, the communication graph of a concurrent program depends on how the communications are ordered on each process. In this article, this point is addressed as follows. We build a finite state automaton $\widehat{\mathcal{A}}$ which indicates how the synchronization points possibly follow each other. For each labeled sub-expression e^l in the program, we use an automaton whose initial and final states are identified by the nodes B_l and E_l. The transitions

between these nodes depends on the expression. The transitions between sub-expressions indicate their respective evaluation order. A transition related to a sequential reduction step is labeled ε and we introduce a l-transition to denote the occurrence of a synchronization point labeled l. The result is an approximation of the communication scheme of each process.

– Knowing how the synchronizations are ordered on each process, we have to determine their interactions. A first way to address this problem is to consider the product of the automata related to the different processes in the program. Based on the collection $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}}_p)_{p \in \mathrm{Dom}(P)}$ of automata describing the synchronization points of the different processes, we show that the product automaton $\mathrm{A}^{\natural}_{\otimes}(\widehat{\mathcal{S}})$ describes a correct approximation of the synchronizations the program may realize. However, on one hand the size of $\mathrm{A}^{\natural}_{\otimes}(\widehat{\mathcal{S}})$ is possibly exponential in the size of the program, while on the other hand it contains more informations than really needed for the analysis. Intuitively, the only relevant informations in $\mathrm{A}^{\natural}_{\otimes}(\widehat{\mathcal{S}})$ for the analysis are the points which may synchronize together. Hence, we introduce a reduced product automaton $\mathrm{A}^{\sharp}_{\otimes}(\widehat{\mathcal{S}})$ which is polynomial in the size of the source program and we prove that $\mathrm{A}^{\sharp}_{\otimes}(\widehat{\mathcal{S}})$ correctly approximates $\mathrm{A}^{\natural}_{\otimes}(\widehat{\mathcal{S}})$. The result is an approximation of the possible interactions between processes.

In Figure 1 c), we show the approximation of the topology obtained using the method described above for the communication scheme of Figure 1 a). In the remainder of this Section we discuss the approximations made by our CFA.

First, the abstract value attached to a sub-expression is the union of all the abstract values carried by the different reduction paths of a non-deterministic program. For communications, this means that the value attached to a reception is the union of all the values carried by the potential emitters. This corresponds to the approximation done in Figure 1 c). Such approximations make the control flow annotations grow. However, any abstract value collected this way is related to one of the concrete values in one of the possible communication schemes.

Other approximations are obviously done during the analysis of the body of loops. First, different channel names created by the same `channel()` instruction inside a loop are identified. Hence, communications over such channels are assumed to be possible, even though the emitter and the receptor use different instances of the same instruction.

Second, a process p may communicate with the communication points occurring inside the body b of a loop, as long as p assumes that the loop has not terminated. However, thanks to the sequential automata, the analysis keeps track of the communication ordering inside b. The only sequences allowed are those corresponding to an unfolding of b.

For instance, let us consider the program of Figure 2 which describes a multiplexer-demultiplexer of channels. This program is made of two processes p_1 and p_2 written in Concurrent ML. Each sub-expression is labeled by an integer. p_1 alternatively receives data on channels ι_1 and ι_2. These data are transmitted on a channel γ. p_2 receives on γ the multiplexed data and transmits them on o_1 and o_2. We assume that γ is only shared by p_1 and p_2. So, the data sent on o_1

(resp. o_2) are the ones the multiplexer received on ι_1 (resp. ι_2). Finally, a process p_3 sends new channel names to the multiplexer on ι_1 and ι_2.

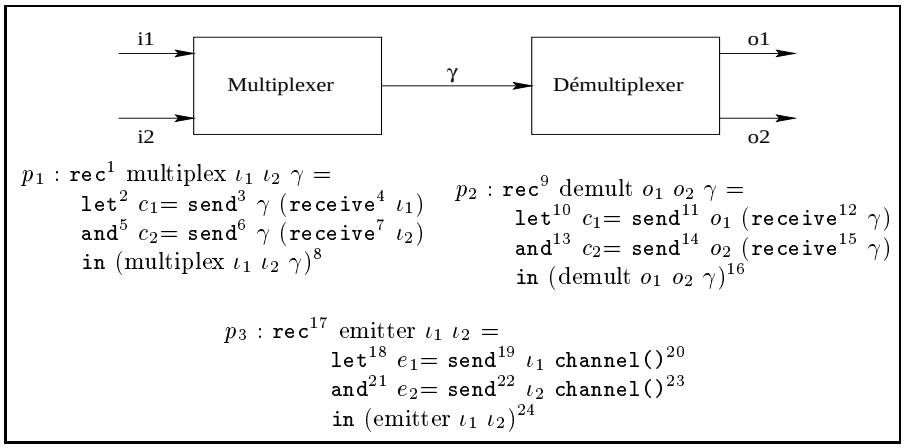


Fig. 2. Multiplexer - demultiplexer written in Concurrent ML.

Our implementation of the analysis determines that the abstract value emitted at point 11 (resp. 14) is the singleton $\{20\}$ (resp. $\{23\}$). As stated before, different channel names created at the same point are not distinguished. However, the CFA detects that the instance of the first reception on γ (point 12) only may receive a value sent by an instance of the first emission (point 3). Similar observations can be done for the second emission and reception on γ.

3 Concurrent ML

In this Section, we introduce the subset of Concurrent ML [3, 20, 21] used in this article to illustrate how our analysis works. The syntax is defined in Figure 3.

Labels are attached to terms. They are used during the analysis which assumes that each sub-expression in the initial program has a unique label. This property is not maintained by reduction. The language contains conditionals, a `let` construct, and the operator `rec` for recursive functions. $\texttt{channel()}^l$ denotes a function call which creates and returns a new channel name k different from all the existing ones. $(\texttt{fork}\ e_0^{l_0})^l$ creates a new process which computes $e_0^{l_0}$ and evaluates to `()`(the value of type unit). $(\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^l$ is the blocking emission of the value of $e_1^{l_1}$ on the channel resulting from the evaluation of $e_0^{l_0}$. $e_0^{l_0}$ and $e_1^{l_1}$ respectively are the *subject* and the *object* of the communication. Once the communication is done, a send evaluates to the value of its object. $(\texttt{receive}\ e_0^{l_0})^l$ is the blocking reception of a value on the channel name described by $e_0^{l_0}$ (the subject of the reception). Values are in the domains of basic types or channel names or functions.

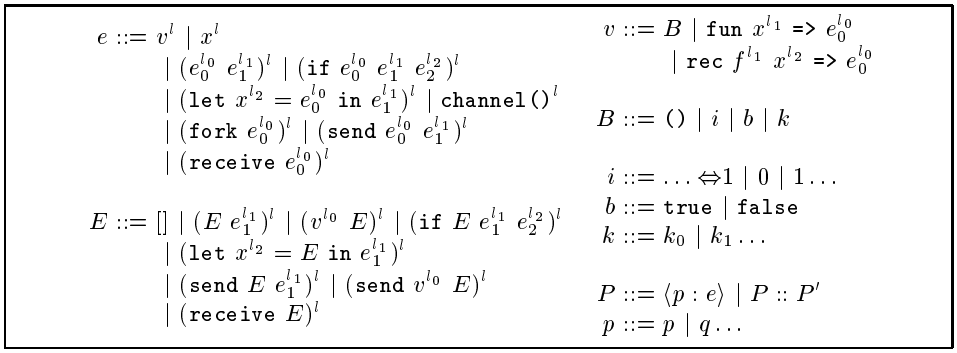


Fig. 3. Language definition.

The operational semantics, given in Figure 4, are based on the language λ_{cv} defined by Reppy [20]. $\hookrightarrow$ is used for sequential reduction steps. $e_0^{l_0}\{x \leftarrow e_1^{l_1}\}$ denotes the term obtained by discarding the labels of the occurrences of x in $e_0^{l_0}$ and substituting $e_1^{l_1}$ to x.

$\xrightarrow{[\ell]}$ is used for concurrent reduction steps. These steps are annotated with labels $\ell \in \mathrm{Lab}^2 \cup \{\varepsilon\}$ which only are used in order to prove the correctness of the analysis. An ε-labeled step corresponds to a sequential reduction step made by one of the processes in the pool. The transition related to a communication between two instructions labeled l_s and l_r is annotated l_s, l_r and the reduction step related to a process creation is annotated l_f, l_f where l_f is the label of the `fork`. Following Reppy [20], we use evaluation contexts E defined in Figure 3.

In addition, we use the classical notion of *configuration* K, P to denote a concurrent system in Concurrent ML. K is the environment for channels and P, the *process pool*, is defined in Figure 3.

A process is defined by $\langle p : e^l \rangle$, p being the name associated to the process which computes e^l. In Figure 3, $\xrightarrow{[c]}$ is defined over process pools. Inside a process, a sequential reduction step is done using $\hookrightarrow$.

For the instruction $(\texttt{channel()})^l$, a new channel name k is assigned to the expression and the set K of used channel names is enriched with k. When an expression is forked, the father evaluates to `()` and the new process is named q, where q is a fresh process name. Finally, a communication may occur between an emitting process p_s and a receiving process p_r which use the same channel. In this case, both p_s and p_r evaluate to the value exchanged.

4 Analysis of the Sequential Expressions

In this Section, we introduce the basic analysis within a sequential process. Section 5 presents the analysis between processes. Note that there are not separable analyses, because the interactions between processes affect the control-flow analysis of the individual processes. If e^l contains `fork`'s, the bodies of the child

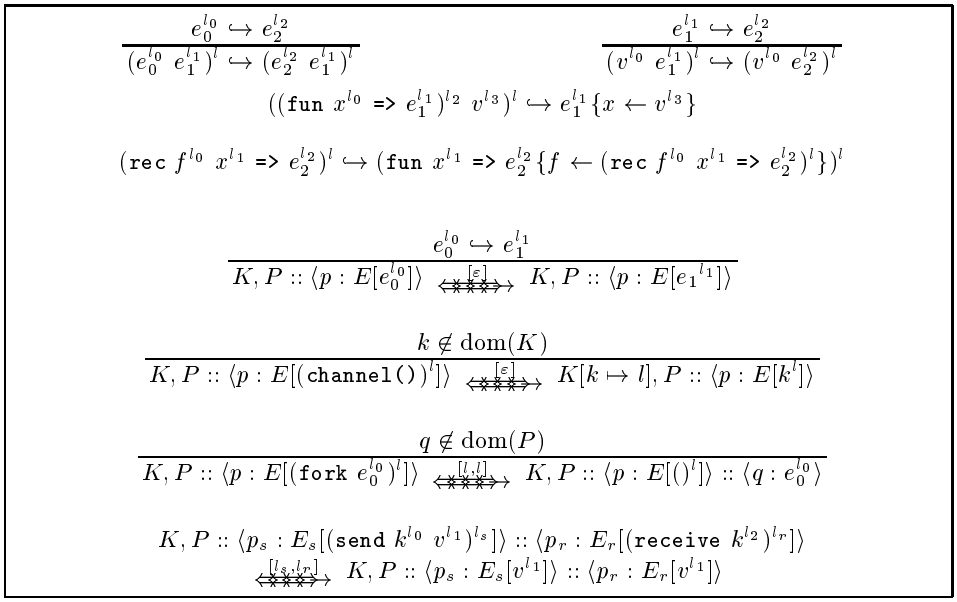


Fig. 4. Language semantics.

processes are analyzed at the same time as e^l, but not their interactions whose analysis is deferred to Section 5.

The analysis of an expression e^l is a triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}})$. Let Lab and Id respectively denote the sets of labels and variables occurring in e^l. $\widehat{\mathcal{C}}$, $\widehat{\mathcal{E}}$ and $\widehat{\mathcal{A}}$ are defined as follows.

- $\widehat{\mathcal{C}}$: $\mathrm{Lab} \to \wp(\mathrm{Lab})$ is the abstract cache which approximates, for any subexpression e^l in the program, the set $\widehat{\mathcal{C}}(l)$ of values e^l may evaluate to. The abstract values either are functions denoted by their labels or channel names denoted by the label of the instruction `channel()` which created them. Type discipline ensures that no confusion is made between both kinds of values and avoids the introduction of two different caches.
- $\widehat{\mathcal{E}}$: $\mathrm{Id} \to \wp(\mathrm{Lab})$ is the abstract environment which binds free variables during the analysis.
- $\widehat{\mathcal{A}} = (\Sigma, \mathrm{B}, Q, Q_f, \delta)$ is a finite automaton which indicates how the synchronizations are ordered inside the expression we analyze. $\Sigma = \mathrm{Lab} \cup \{\varepsilon, \mu\}$ is the alphabet, Q is the set of state, $\mathrm{B} \in Q$ is the initial state, $Q_f = \{\mathrm{E}\} \subseteq Q$ contains a unique final state. $\delta \in (Q \times \Sigma) \to \wp(Q)$ is the transition function.

In Figure 5, we define inductively on the structure of the terms the constraints a triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}})$ has to satisfy in order to define a correct analysis for an expression e^l. If so, we write $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}} \vdash e^l$.

We use the following notations. $[\![\mathrm{s} \xrightarrow{\ell} \mathrm{s}']\!]$ denotes an automaton whose initial and final state respectively are s and s′ and such that $Q = \{\mathrm{s}, \mathrm{s}'\}$ and $\mathrm{s}' \in \delta(\mathrm{s}, \ell)$.

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash c^l \Leftrightarrow [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \qquad \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash x^l \Leftrightarrow \widehat{\mathcal{E}}(x) \subseteq \widehat{\mathcal{C}}(l),\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (\texttt{fun}\ x^{l_1}\ \texttt{=>}\ e_0^{l_0})^l \Leftrightarrow \begin{cases} l \in \widehat{\mathcal{C}}(l),\ \widehat{\mathcal{C}},\widehat{\mathcal{E}}[x \mapsto \widehat{\mathcal{C}}(l_1)],\widehat{\mathcal{A}}_0 \vdash e_0^{l_0},\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}} \\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{\mu} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_0} \xrightarrow{\mu} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \end{cases}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (\texttt{rec}\ f^{l_1}\ x^{l_2}\ \texttt{=>}\ e_0^{l_0})^l \Leftrightarrow \begin{cases} l \in \widehat{\mathcal{C}}(l),\ \widehat{\mathcal{C}}(l) \subseteq \widehat{\mathcal{C}}(l_1),\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}} \\ \widehat{\mathcal{C}},\widehat{\mathcal{E}}[f \mapsto \widehat{\mathcal{C}}(l_1)][x \mapsto \widehat{\mathcal{C}}(l_2)],\widehat{\mathcal{A}}_0 \vdash e_0^{l_0} \\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{\mu} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_0} \xrightarrow{\mu} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \end{cases}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (e_0^{l_0}\ e_1^{l_1})^l \Leftrightarrow \begin{cases} \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_0 \vdash e_0^{l_0},\ \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_1 \vdash e_1^{l_1} \\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}},\ \widehat{\mathcal{A}}_1 \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_0} \xrightarrow{\varepsilon} \mathrm{B}_{l_1}]\!] \sqsubseteq \widehat{\mathcal{A}} \\ \forall l_2 \in \widehat{\mathcal{C}}(l_0)\ :\ (\texttt{fun}\ x^{l_4}\ \texttt{=>}\ e_3^{l_3})^{l_2} \in \mathrm{prg}, \\ \quad \widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l_4),\ \widehat{\mathcal{C}}(l_3) \subseteq \widehat{\mathcal{C}}(l), [\![\mathrm{E}_{l_1} \xrightarrow{\varepsilon} \mathrm{B}_{l_3}]\!] \sqsubseteq \widehat{\mathcal{A}}, [\![\mathrm{E}_{l_3} \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \\ \forall l_2 \in \widehat{\mathcal{C}}(l_0)\ :\ (\texttt{rec}\ f^{l_4}\ x^{l_5}\ \texttt{=>}\ e_3^{l_3})^{l_2} \in \mathrm{prg}, \\ \quad \widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l_5),\ \widehat{\mathcal{C}}(l_3) \subseteq \widehat{\mathcal{C}}(l) \\ \quad [\![\mathrm{E}_{l_1} \xrightarrow{\varepsilon} \mathrm{B}_{l_3}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_3} \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \\ \quad [\![\mathrm{E}_{l_1} \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_3} \xrightarrow{\varepsilon} \mathrm{B}_{l_3}]\!] \sqsubseteq \widehat{\mathcal{A}} \end{cases}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash \texttt{channel}\,()^{\,l} \Leftrightarrow l \in \widehat{\mathcal{C}}(l),\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash k^l \Leftrightarrow l \in \widehat{\mathcal{C}}(l),\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (\texttt{fork}\ e_0^{l_0})^l \Leftrightarrow \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_0 \vdash e_0^{l_0},\ [\![\mathrm{B}_l \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}},\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{l} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^l \Leftrightarrow \begin{cases} \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_0 \vdash e_0^{l_0},\ \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_1 \vdash e_1^{l_1},\ \widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l) \\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}},\ \widehat{\mathcal{A}}_1 \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}} \\ [\![\mathrm{E}_{l_0} \xrightarrow{\varepsilon} \mathrm{B}_{l_1}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_1} \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \end{cases}$$

$$\widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}} \vdash (\texttt{receive}\ e_0^{l_0})^l \Leftrightarrow \begin{cases} \widehat{\mathcal{C}},\widehat{\mathcal{E}},\widehat{\mathcal{A}}_0 \vdash e_0^{l_0} \\ \widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}},\ [\![\mathrm{E}_{l_0} \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}} \end{cases}$$

Fig. 5. Specification of the analysis.

$\widehat{\mathcal{A}} \sqsubseteq \widehat{\mathcal{A}}'$ states that $\widehat{\mathcal{A}}$ is a sub-automaton of $\widehat{\mathcal{A}}'$, i.e. $Q \subseteq Q'$ and $\delta \subseteq \delta'$. Hence, $[\![\mathrm{S} \xrightarrow{\ell} \mathrm{S}']\!] \sqsubseteq \widehat{\mathcal{A}}$ indicates that there is a ℓ-transition from S to S′ in $\widehat{\mathcal{A}}$. The analysis builds for each expression e^l an automaton $\widehat{\mathcal{A}}$ which initial and final states respectively are $\mathrm{Start}(\widehat{\mathcal{A}}) = \mathrm{B}_l$ and $\mathrm{End}(\widehat{\mathcal{A}}) = \{\mathrm{E}_l\}$. These automata are defined inductively on the structure of the terms. Their graphical representation is given in Figure 6. ε-transitions denote sequential reduction steps.

Because the evaluation of a first order constant c^l does not involve any synchronization, the automaton associated to this expression is made of an ε-transition from the initial state B_l to the final state E_l, as shown in Figure 6. This is ensured by the constraint $[\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$ in the specification of Figure 5.

For a variable x^l, $\widehat{\mathcal{E}}(x)$ is added to $\widehat{\mathcal{C}}(l)$ indicating that the abstract value of point l depends on the abstract value of x in the environment. In addition,

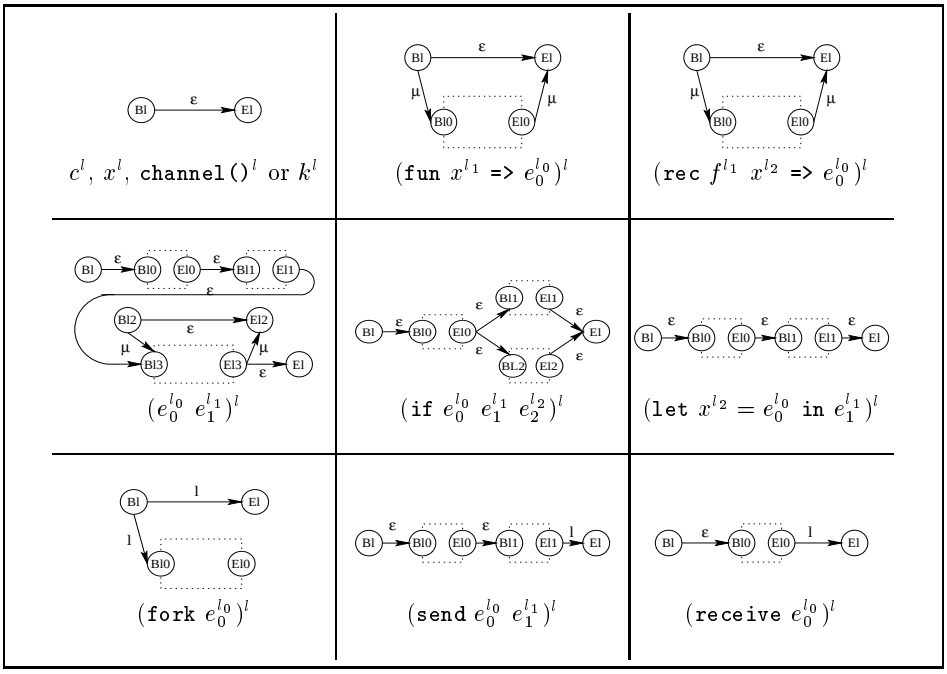


Fig. 6. Automata attached to the expressions during the analysis.

because no reduction may come from the evaluation of a variable, the automaton related to x^l is made of an ε-transition between the states B_l and E_l. Again, this is ensured by the constraint $[\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$.

(fun x^{l_1} => $e_0^{l_0}$)^l and (rec f x^{l_1} => $e_0^{l_0}$)^l are values. Their evaluation does not require any computation. Hence, just like for first order constants, the initial state B_l of the automaton is linked to the final state E_l by an ε-transition. In addition, the body $e_0^{l_0}$ is analyzed, yielding an automaton $\widehat{\mathcal{A}}_0$ with B_{l_0} and E_{l_0} as initial and final states. As depicted in Figure 6, we connect $\widehat{\mathcal{A}}_0$ to $\widehat{\mathcal{A}}$ by a μ-transitions *via* the constraints $[\![\mathrm{B}_l \xrightarrow{\mu} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}$ and $[\![\mathrm{E}_{l_0} \xrightarrow{\mu} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$. μ-transitions only are used to link both parts of the automaton and never describe a valid path between nodes. Concerning the abstract cache, l is added to $\widehat{\mathcal{C}}(l)$, indicating that the current function is among the ones which may occur at this point.

The analysis of an application ($e_0^{l_0}$ $e_1^{l_1}$)^l first relies on these of $e_0^{l_0}$ and $e_1^{l_1}$ which yield two automata $\widehat{\mathcal{A}}_0$ and $\widehat{\mathcal{A}}_1$. Because of the evaluation order, the synchronizations made by $e_0^{l_0}$ precede the ones made by $e_1^{l_1}$. Also, assuming that $e_0^{l_0}$ evaluates to (fun x^{l_4} => $e_3^{l_3}$)^{l_2}, the synchronizations made by $e_1^{l_1}$ precede the ones resulting from the evaluation of $e_3^{l_3}$ with the right value for x. Hence we build the following automaton. B_l is linked to the initial state of $\widehat{\mathcal{A}}_0$ and $\widehat{\mathcal{A}}_0$'s final state is linked to $\widehat{\mathcal{A}}_1$ initial state by ε-transitions. This corresponds to the constraints $\widehat{\mathcal{A}}_0 \sqsubseteq \widehat{\mathcal{A}}$, $\widehat{\mathcal{A}}_1 \sqsubseteq \widehat{\mathcal{A}}$, $[\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}$ and $[\![\mathrm{E}_{l_0} \xrightarrow{\varepsilon} \mathrm{B}_{l_1}]\!] \sqsubseteq \widehat{\mathcal{A}}$.

Next, $\widehat{\mathcal{C}}(l_0)$ denotes the labels of the functions $e_0^{l_0}$ may evaluate to. For each function $(\texttt{fun}\ x^{l_4}\ \texttt{=>}\ e_3^{l_3})^{l_2}$ such that $l_2 \in \widehat{\mathcal{C}}(l_0)$, we indicate that the synchronizations in the body $e_3^{l_3}$ follow these in $e_1^{l_1}$ by asking $[\![\mathrm{E}_{l_1} \xrightarrow{\varepsilon} \mathrm{B}_{l_3}]\!] \sqsubseteq \widehat{\mathcal{A}}$. Finally, the final states of e^l and $e_3^{l_3}$ are linked by $[\![\mathrm{E}_{l_3} \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$. prg denotes the program for which we compute an analysis. In Figure 6 we show the automaton resulting from the analysis of an expression $(e_0^{l_0}\ e_1^{l_1})^l$, assuming that $\widehat{\mathcal{C}}(l_0) = \{l_2\}$ and that $(\texttt{fun}\ x^{l_4}\ \texttt{=>}\ e_3^{l_3})^{l_2} \in \mathrm{prg}$.

The application of a recursive function is analyzed similarly. Because the body may be executed zero or many times, we add $[\![\mathrm{B}_l \xrightarrow{\varepsilon} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$ and $[\![\mathrm{E}_{l_4} \xrightarrow{\varepsilon} \mathrm{B}_{l_4}]\!] \sqsubseteq \widehat{\mathcal{A}}$, where l_4 is the label of the body of the recursive function.

Channels are identified by their creation points. `channel()` being a function call, for an occurrence of $\texttt{channel()}^l$ in the program, we collect l in $\widehat{\mathcal{C}}(l)$. So, no distinction is made between different channels created in a recursive function.

For an expression $(\texttt{fork}\ e_0^{l_0})^l$, we only analyze the body $e_0^{l_0}$ of the child process at this stage of the analysis, obtaining an automaton $\widehat{\mathcal{A}}_0$. In Section 5, during the analysis of the interactions between processes, we consider `fork`'s just like another communications. For now, $[\![\mathrm{B}_l \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$ indicates that the execution of the `fork` consists of doing a synchronization denoted l. On the other hand, the automaton $\widehat{\mathcal{A}}_0$ which describes the synchronizations realized by $e_0^{l_0}$ is included in $\widehat{\mathcal{A}}$ and linked to the initial state B_l of the expression we analyze by $[\![\mathrm{B}_l \xrightarrow{l} \mathrm{B}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}$. In Section 5.2, when considering the product automaton, $e_0^{l_0}$ stays frozen until it can synchronize on l with the `fork`.

The specification of the analysis for an emission $(\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^l$ states that the sub-expressions are analyzed. $[\![\mathrm{E}_{l_0} \xrightarrow{\varepsilon} \mathrm{E}_{l_1}]\!] \sqsubseteq \widehat{\mathcal{A}}$ indicates that the execution of $e_0^{l_0}$ precedes the one of $e_1^{l_1}$ and $[\![\mathrm{E}_{l_1} \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$ states that l denotes the last synchronization point in $(\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^l$.

Receptions are treated similarly. $[\![\mathrm{E}_{l_0} \xrightarrow{l} \mathrm{E}_l]\!] \sqsubseteq \widehat{\mathcal{A}}$ states that the synchronizations in the sub-expression precede the synchronization related to the reception. Note that no constraint is introduced on the received abstract value at this stage of the analysis. This is due to the fact that $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}} \vdash e^l$ only specifies the analysis of the different processes without considering their synchronizations. However, in a concurrent execution, a reception $(\texttt{receive}\ e_0^{l_0})^l$ may receive either a basic value or a function or a name, depending on its type. Hence, in Sections 5.1 and 5.2, when considering process interactions, we introduce constraints indicating how $\widehat{\mathcal{C}}(l)$ is bound to. These constraints are specified in definitions 4 and 8.

As an intermediary result, we introduce some properties satisfied by the analysis defined in Figure 5. First we examine the existence of a best analysis for an expression e^l and, second, we focus on correctness. These results are used further in this article, when proving the main property concerning the correctness of the specification over concurrent reduction.

Let $(\widehat{\mathcal{C}}_1, \widehat{\mathcal{E}}_1, \widehat{\mathcal{A}}_1)$ and $(\widehat{\mathcal{C}}_2, \widehat{\mathcal{E}}_2, \widehat{\mathcal{A}}_2)$ be two analyses for the same expression e^l. $(\widehat{\mathcal{C}}_1, \widehat{\mathcal{E}}_1, \widehat{\mathcal{A}}_1) \prec (\widehat{\mathcal{C}}_2, \widehat{\mathcal{E}}_2, \widehat{\mathcal{A}}_2)$ denotes that $(\widehat{\mathcal{C}}_1, \widehat{\mathcal{E}}_1, \widehat{\mathcal{A}}_1)$ is more precise than

$(\widehat{\mathcal{C}}_2, \widehat{\mathcal{E}}_2, \widehat{\mathcal{A}}_2)$. $\prec$ is defined by

$$(\widehat{\mathcal{C}}_1, \widehat{\mathcal{E}}_1, \widehat{\mathcal{A}}_1) \prec (\widehat{\mathcal{C}}_2, \widehat{\mathcal{E}}_2, \widehat{\mathcal{A}}_2) \Leftrightarrow \begin{cases} \forall l \in \text{Lab},\ \widehat{\mathcal{C}}_1(l) \subseteq \widehat{\mathcal{C}}_2(l) \\ \forall x \in \text{Id},\ \widehat{\mathcal{E}}_1(x) \subseteq \widehat{\mathcal{E}}_2(x) \\ \widehat{\mathcal{A}}_1 \sqsubseteq \widehat{\mathcal{A}}_2 \end{cases} \quad (2)$$

The existence of a least analysis in the sense of $\prec$ stems from the fact that, for an expression e^l, the set $\{(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}) \ :\ \widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}} \vdash e^l\}$ is a Moore family[1].

The main property introduced in this Section concerns sequential subject reduction and indicates that if a triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}})$ is a correct analysis for an expression e^l then it is still correct after a sequential reduction step $\hookrightarrow$ (Section 3). First, we introduce the order relation $\lessdot$ over automata.

Definition 1 *Let $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{A}}'$ be two automata. $\widehat{\mathcal{A}} \lessdot \widehat{\mathcal{A}}'$ iff any path labeled $\ell_1 \ldots \ell_n$ in $\widehat{\mathcal{A}}$ is a path in $\widehat{\mathcal{A}}'$.*

Intuitively, the labels of the transitions in an automaton $\widehat{\mathcal{A}}$ built during the analysis of an expression e^l correspond to the synchronizations realized during the execution of e^l. Hence, a path in $\widehat{\mathcal{A}}$ denotes one possible sequence of synchronizations the execution of e^l may lead to. Sequential reduction steps do not realize synchronizations but may discard some possible sequences of synchronizations. For instance, consider the execution of a conditional. Hence, if $e^l \hookrightarrow e'^{l'}$ then any possible sequence of communication in $e'^{l'}$ is a sequence of communication in e^l. Considering the automata $\widehat{\mathcal{A}}$ and $\widehat{\mathcal{A}}'$ built during the analysis for e^l and $e'^{l'}$, we have $\widehat{\mathcal{A}}' \lessdot \widehat{\mathcal{A}}$. This is summed up by the proposition below.

Proposition 2 (Sequential subject reduction) *If $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}} \vdash e^l$ and $e^l \hookrightarrow e'^{l'}$ then $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}' \vdash e'^{l'}$ for some $\widehat{\mathcal{A}}'$ such that $\widehat{\mathcal{A}}' \lessdot \widehat{\mathcal{A}}$.*

Analyzing separately the processes of an application allows us to order the synchronization points on each process. However this is not enough to obtain a fine approximation of the topology of communications. In the following Section, we consider the product automaton of the automata described above in order to compute the synchronization realized by the program.

5 Process Pool Analysis

In this Section, we focus on analyzing a process pool. Section 5.1 defines an analysis $\models^\natural$ based on the product automaton $\text{A}^\natural_\otimes(\widehat{\mathcal{S}})$ of a collection $\widehat{\mathcal{S}}$ of automata built as in Section 4. We prove the correctness of $\models^\natural$ by a subject reduction property. In Section 5.2, we introduce a second analysis $\models^\sharp$ based on a reduced product automaton $\text{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ which size is polynomial in the size of the original program. We prove that $\models^\sharp$ is a correct approximation of $\models^\natural$.

[1] A subset X of a complete lattice $(L, \leq)$ is a Moore family iff for all set $Y \subseteq X$, $(\sqcap Y) \in X$. Notice that a Moore family X never is empty and admits a least element since $(\sqcap X) \in X$ (see [19]).

5.1 Product Automaton Based Analysis

Let $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}}_p)_{p\in \mathrm{Dom}(P)}$ be a collection of automata. The states in the product automaton $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ are products of the states of the automata in $\widehat{\mathcal{S}}$. An ε-transition inside an automaton $\widehat{\mathcal{A}}_p$ denotes a sequential reduction step and is transcribed in $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$. An l-transition in $\widehat{\mathcal{A}}_p$ denotes a synchronization point. We add (l,l')-transitions in $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ if l and l' are transitions in two automata of $\widehat{\mathcal{S}}$ such that the instruction labeled l may synchronize with the instruction labeled l'.

Definition 3 (Product automaton) *Let $\widehat{\mathcal{S}}$ be a collection of n automata. The product automaton $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ of the automata in $\widehat{\mathcal{S}}$ is a tuple $(\Sigma^\natural, q_0^\natural, Q^\natural, \delta^\natural)$. The alphabet is $\Sigma^\natural \subseteq Lab^2 \cup \{\varepsilon, \mu\}$. $Q^\natural$ is made of k-tuples $(\mathrm{s}_1, \ldots, \mathrm{s}_k)$ in which s_i, $1 \leq i \leq k$, denotes the advancement of the i^{th} automaton. $q_0^\natural$ is the initial state and $\delta^\natural \in (Q^\natural \times \Sigma^\natural) \to \wp(Q^\natural)$ is the transition function. $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ is built as follows.*

(i) The initial state $q_0^\natural \in Q^\natural$ is the product of the initial states $Start(\widehat{\mathcal{A}}_p)$ of the automata $\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}$, i.e.

$$q_0^\natural \stackrel{def}{=} \bigotimes_{\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}} Start(\widehat{\mathcal{A}}_p) \tag{3}$$

(ii) For all $q^\natural \in Q^\natural$ such that $q^\natural = (\mathrm{s}_0, \ldots, \mathrm{s}_n)$,

(a) $\forall i,\ 1 \leq i \leq n$, $\forall \ell \in \{\varepsilon, \mu\}$,
$$\left| \begin{array}{l} \left(\exists \widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}} \ : \ [\![\mathrm{s}_i \stackrel{\ell}{\rightharpoonup} \mathrm{s}_i']\!] \sqsubseteq \widehat{\mathcal{A}}_p\right) \ \Rightarrow \\ [\![q^\natural \stackrel{\ell}{\rightharpoonup} (\mathrm{s}_1, \ldots, \mathrm{s}_i', \ldots, \mathrm{s}_n)]\!] \sqsubseteq \mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}}) \end{array} \right.$$

(b) $\forall i, j,\ 1 \leq i \neq j \leq n$,

$$\left(\exists \widehat{\mathcal{A}}_p, \widehat{\mathcal{A}}_{p'} \in \widehat{\mathcal{S}} \ : \ \left| \begin{array}{l} [\![\mathrm{s}_i \stackrel{l_s}{\rightharpoonup} \mathrm{s}_{l_s}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ [\![\mathrm{s}_j \stackrel{l_r}{\rightharpoonup} \mathrm{s}_{l_r}]\!] \sqsubseteq \widehat{\mathcal{A}}_{p'} \\ (\textbf{\textit{send}}\ e_0^{l_0}\ e_1^{l_1})^{l_s} \in prg \\ (\textbf{\textit{receive}}\ e_2^{l_2})^{l_r} \in prg \end{array} \right. \right) \Rightarrow [\![q^\natural \stackrel{l_s, l_r}{\rightharpoonup} (\mathrm{s}_1, \ldots, \mathrm{s}_{l_s}, \ldots, \mathrm{s}_{l_r}, \ldots, \mathrm{s}_n)]\!] \sqsubseteq \mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}}) \tag{4}$$

(c)

$$\forall i, 1 \leq i \leq n, \left| \begin{array}{l} \left(\exists \widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}} \ : \ \left| \begin{array}{l} [\![\mathrm{s}_i \stackrel{l_f}{\rightharpoonup} \mathrm{s}_{l_f}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ [\![\mathrm{s}_i \stackrel{l_f}{\rightharpoonup} \mathrm{s}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ (\textbf{\textit{fork}}\ e_0^{l_0})^{l_f} \in prg \end{array} \right. \right) \\ \Rightarrow [\![q^\natural \stackrel{l_f, l_f}{\rightharpoonup} (\mathrm{s}_1, \ldots, \mathrm{s}_{l_f}, \ldots, \mathrm{s}_n, \mathrm{s}_{l_0})]\!] \sqsubseteq \mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}}) \end{array} \right. \tag{5}$$

The product automaton is built incrementally. We start with the only state $q_0^\natural$ and any state $q^\natural = (\mathrm{s}_1, \ldots, \mathrm{s}_k)$ in $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ describes one of the possible advancements of the process pool. We add an ε-transition going out of $q^\natural$ every-time there is an ε-transition in one of the automata $\widehat{\mathcal{A}}_p$ going out of s_i, $1 \leq i \leq k$. Doing so, we obtain a new state which also denotes one of the possible advancements.

Next, an l-transition denotes a synchronization point. Two processes may communicate together if they own matching synchronization points l_s and l_r

related to an emission and a reception and which possibly are active at the same time. In this case there is a state $q^\sharp = (\mathrm{s}_1, \ldots, \mathrm{s}_k)$ and two automata $\widehat{\mathcal{A}}_p$ and $\widehat{\mathcal{A}}_{p'}$ in $\widehat{\mathcal{S}}$ such that $[\![\mathrm{s}_s \xrightarrow{l_s} \mathrm{s}'_s]\!] \sqsubseteq \widehat{\mathcal{A}}_p$ and $[\![\mathrm{s}_r \xrightarrow{l_r} \mathrm{s}'_r]\!] \sqsubseteq \widehat{\mathcal{A}}_{p'}$, $1 \leq s, r \leq k$, $s \neq r$. Then the transition $q^\sharp \xrightarrow{l_s, l_r} q^{\sharp\prime}$ is added to the product automaton where $q^{\sharp\prime}$ is a new state obtained by substituting s'_s and s'_r to s_s and s_r in $q^\sharp$.

Notice that the product automaton allows a communication between any emitter s and any receptor r which possibly are active at the same time, independently of the channel they use. In Definition 4, the abstract value sent by s is added to the abstract value attached to r iff s and r possibly communicate on the same channel. Greater precision would be obtained by directly discarding these impossible communications in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$. However, the product automaton would not be defined independently of the analysis, overloading the notations.

Finally, a `fork` creates a new process. Hence, for any transition $[\![\mathrm{s}_i \xrightarrow{l_f} \mathrm{s}_{l_f}]\!] \sqsubseteq \widehat{\mathcal{A}}_p$ such that there exists a state $q^\sharp = (\mathrm{s}_1, \ldots, \mathrm{s}_i, \ldots, \mathrm{s}_k)$ in $Q^\sharp$, the $(k+1)$ product state $(\mathrm{s}_1, \ldots, \mathrm{s}_{l_f}, \ldots, \mathrm{s}_k, \mathrm{s}_{l_0})$ is added to $Q^\sharp$. This tuple denotes the state resulting from the execution of the `fork`. It contains one more component s_{l_0} which describes the advancement of the new process. Concerning the father process, the state denoting its advancement is updated to indicate that the process creation is done.

In the remainder of this Section, we specify the conditions a triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes)$ has to satisfy to be an analysis for a process pool P and we prove the correctness of this analysis. $\widehat{\mathcal{C}}$ and $\widehat{\mathcal{E}}$ are defined in the same way as in Section 4 and $\widehat{\mathcal{A}}^\sharp_\otimes$ is the product of the automata $\widehat{\mathcal{A}}_p$ built for the sequential expressions of P. Also, we introduce a new order relation extending $\preccurlyeq$ to deal with the way the product automata are related under concurrent reduction.

$\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes \models^\sharp P$ denotes that the triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes)$ is an analysis for the process pool P, where $\models^\sharp$ is specified in Definition 4. Intuitively, we require that all the expressions in the process pool are correctly abstracted using the automaton $\widehat{\mathcal{A}}^\sharp_\otimes$ and that for any communication described in $\widehat{\mathcal{A}}^\sharp_\otimes$, the abstract value sent by the emitter is contained in the abstract value attached to the receptor.

Definition 4 *A triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes)$ defines good annotations for a process pool P, denoted $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes \models^\sharp P$ iff*

(i) $\widehat{\mathcal{A}}^\sharp_\otimes = \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ *for some collection* $\widehat{\mathcal{S}}$ *of automata such that* $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}}_p)_{p \in Dom(P)}$ *and* $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}_p \vdash e^l$ *for all* $\langle p : e^l \rangle \in P$.

(ii) For all $[\![q \xrightarrow{l_s, l_r} q']\!] \sqsubseteq \widehat{\mathcal{A}}^\sharp_\otimes$ *s.t.* $\begin{cases} (\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^{l'_i} \in prg \\ (\texttt{receive}\ e_2^{l_2})^{l'_j} \in prg \end{cases}$, $\widehat{\mathcal{C}}(l_0) \cap \widehat{\mathcal{C}}(l_2) \neq \varnothing$ *implies* $\widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l_r)$.

As stated in Section 4, the abstract cache $\widehat{\mathcal{C}}$ indifferently is used to collect abstract values denoting either channels or functions. In Definition 4, $e_0^{l_0}$ and $e_2^{l_2}$ have type Channel and consequently the condition $\widehat{\mathcal{C}}(l_0) \cap \widehat{\mathcal{C}}(l_2) \neq \varnothing$ considers abstract values related to channels in order to determine whether the emitter

and the receptor may communicate. If so, the constraint $\widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l_r)$ involves abstract values denoting channels or functions, depending on the type of $e_1^{l_1}$.

Similarly to Section 4, the existence of a least analysis for a process pool P stems from the fact that the set $\{(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\natural_\otimes) \ : \ \widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\natural_\otimes \models^\natural P\}$ of correct analysis for a process pool P is a Moore family w.r.t. the relation $\prec$ of Equation (2).

In order to prove that an analysis is preserved under reduction, we introduce a new order relation $\overset{\ell}{\lessdot}$ which extends $\lessdot$ as follows. A path in $\widehat{\mathcal{A}}^\natural_\otimes$ describes the sequence of synchronizations realized by one possible execution of the program. This relation allows us to state that if $K, P \xrightarrow{[\ell]} K', P'$ then the product automaton $\widehat{\mathcal{A}}^\natural_\otimes{}'$ which abstracts P' contains all the paths in $\widehat{\mathcal{A}}^\natural_\otimes$ starting from $q^{\natural\prime}$ where $q^{\natural\prime}$ is a state we access from $q_0^\natural$ by an ℓ-transition. So, any sequence of synchronizations in $\widehat{\mathcal{A}}^\natural_\otimes{}'$ is a sequence of synchronization in $\widehat{\mathcal{A}}^\natural_\otimes$ following the synchronization described by ℓ.

Definition 5 *Let $\widehat{\mathcal{A}}^\natural_\otimes$ and $\widehat{\mathcal{A}}^\natural_\otimes{}'$ be two product automata and $\ell \in \Sigma$ a letter in the alphabet. $\widehat{\mathcal{A}}^\natural_\otimes \overset{\ell}{\lessdot} \widehat{\mathcal{A}}^\natural_\otimes{}'$ iff for each path labeled $\ell_1 \dots \ell_n$ in $\widehat{\mathcal{A}}^\natural_\otimes$ there exists a path labeled $\ell.\ell_1 \dots \ell_n$ in $\widehat{\mathcal{A}}^\natural_\otimes{}'$.*

Notice that $\widehat{\mathcal{A}}^\natural_\otimes \overset{\varepsilon}{\lessdot} \widehat{\mathcal{A}}^\natural_\otimes{}'$ iff $\widehat{\mathcal{A}}^\natural_\otimes \lessdot \widehat{\mathcal{A}}^\natural_\otimes{}'$. Finally, we introduce the following property which describes how the annotations behave under reduction.

Proposition 6 (Concurrent subject reduction) *Let P be a process pool such that $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\natural_\otimes \models^\natural P$. Then if $K, P \xrightarrow{[\ell]} K, P'$ then $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\natural_\otimes{}' \models^\natural P'$ for some product automaton $\widehat{\mathcal{A}}^\natural_\otimes{}'$ such that $\widehat{\mathcal{A}}^\natural_\otimes{}' \overset{\ell}{\lessdot} \widehat{\mathcal{A}}^\natural_\otimes$.*

5.2 Polynomial Size Analysis

In this Section, we introduce an automaton $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ which is a reduced version of $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$. While the size of the product automaton is possibly exponential in the size n of the program, $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ has size $O(n^3)$. We show that $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ can be used instead of $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ for the analysis.

Intuitively, the only relevant informations needed by the analysis are the pairs of synchronizations points which may interact together. For instance, we are interested in the set of potential emission points which may communicate with a given reception point, independently of any notion of ordering, i.e. of the place of this communication in a trace of the execution.

The product automaton $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ contains all the sequences of synchronizations of the whole possible executions, which is more precise than needed for our purpose. Hence, we reduce its size by discarding such irrelevant informations and still keeping enough precision to eliminate non possible synchronizations.

This compromise is obtained in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ by building an automaton able to answer to the question: “which are the possible synchronization points possibly

following a given synchronization?". Let l_0 and l_1 be the labels of matching synchronization points. The state q_{l_0,l_1} denotes this synchronization in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ and the set of synchronization points following the interaction is given by $L(q_{l_0,l_1})$.

A new synchronization between points l_2 and l_3 is allowed if both l_2 and l_3 belong to $L(q_{l,l'})$ for some previous synchronization denoted by $q_{l,l'}$. In this case, we add an (l_2,l_3)-transition from $q_{l,l'}$ to q_{l_2,l_3} and $L(q_{l_2,l_3})$ is updated.

Let S denote the set of states terminating an (l_s,l_r)-transition in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$. S is approximated in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ by a single state q_{l_s,l_r}. Hence, any (l_s,l_r)-transition in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ terminates in q_{l_s,l_r}. In addition, $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ keeps track of the advancement of the sequential automata after the synchronization denoted by (l_s,l_r) via $L(q_{l_s,l_r})$ which contains all the nodes s such that s occurs in q for some $q \in S$.

Definition 7 (Reduced product automaton) *Let $\widehat{\mathcal{S}}$ be a collection of n automata. The reduced product automaton $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ of the automata in $\widehat{\mathcal{S}}$ is a tuple $(\Sigma^\sharp, q_0^\sharp, Q^\sharp, \delta^\sharp, L)$. The alphabet is $\Sigma^\sharp = Lab^2$. The states belong to $Q^\sharp = \{q_{l,l'} : l,l' \in Lab\} \cup \{q_0^\sharp\}$ where $q_0^\sharp$ is a fresh initial state. $\delta^\sharp \in (Q^\sharp \times \Sigma^\sharp) \to \wp(Q^\sharp)$ is the transition function. $L : Q^\sharp \to \wp(Q_P)$, where $Q_P = \cup_{\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}} State(\widehat{\mathcal{A}}_p)$, assigns to any state $q_{l,l'}$ the possibly active states of the $\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}$ once the points l and l' have synchronized together. $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ is built as follows.*

(i) $q_0^\sharp \in Q^\sharp$ *and* $L(q_0^\sharp) = \cup_{\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}} Start(\widehat{\mathcal{A}}_p)$.

(ii) For all $q \in Q^\sharp$,

(a) $\forall \mathrm{s} \in L(q), \left(\exists \widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}} : [\![\mathrm{s} \xrightarrow{\varepsilon} \mathrm{s}']\!] \sqsubseteq \widehat{\mathcal{A}}_p\right) \Rightarrow \{\mathrm{s}'\} \subseteq L(q)$

(b) For all $\mathrm{s}_s \in L(q)$, $\mathrm{s}_r \in L(q)$,

$$\left(\exists \widehat{\mathcal{A}}_p, \widehat{\mathcal{A}}_{p'} \in \widehat{\mathcal{S}} : \begin{cases} [\![\mathrm{E}_{l_1} \xrightarrow{l_s} \mathrm{s}_{l_s}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ [\![\mathrm{E}_{l_2} \xrightarrow{l_r} \mathrm{s}_{l_r}]\!] \sqsubseteq \widehat{\mathcal{A}}_{p'} \\ (\textbf{\textit{send}}\ e_0^{l_0}\ e_1^{l_1})^{l_s} \in prg \\ (\textbf{\textit{receive}}\ e_2^{l_2})^{l_r} \in prg \end{cases}\right) \Rightarrow \begin{cases} [\![q \xrightarrow{l_s,l_r} q_{l_s,l_r}]\!] \sqsubseteq \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}}) \\ \Lambda(q) \setminus \{\mathrm{E}_{l_1}, \mathrm{E}_{l_2}\} \subseteq L(q_{l_s,l_r}) \\ \{\mathrm{s}_{l_s}, \mathrm{s}_{l_r}\} \subseteq L(q_{l_s,l_r}) \end{cases} \tag{6}$$

(c) $\forall \mathrm{s}_f \in L(q)$,

$$\left(\exists \widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}} : \begin{cases} [\![\mathrm{B}_{l_f} \xrightarrow{l_f} \mathrm{s}_{l_f}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ [\![\mathrm{B}_{l_f} \xrightarrow{l_f} \mathrm{s}_{l_0}]\!] \sqsubseteq \widehat{\mathcal{A}}_p \\ (\textbf{\textit{fork}}\ e_0^{l_0})^{l_f} \in prg \end{cases}\right) \Rightarrow \begin{cases} [\![q \xrightarrow{l_f,l_f} q_{l_f,l_f}]\!] \sqsubseteq \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}}) \\ L(q) \setminus \{\mathrm{B}_{l_f}\} \subseteq L(\mathrm{s}_{l_f,l_f}) \\ \{\mathrm{s}_{l_f}, \mathrm{s}_{l_0}\} \subseteq L(\mathrm{s}_{l_f,l_f}) \end{cases} \tag{7}$$

$\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ is built incrementally. We start with a fresh initial state $q_0^\sharp$. $L(q_0^\sharp)$ contains the initial states of the automata in $\widehat{\mathcal{S}}$ indicating that the related points belong to the set of possible active points at the beginning of the execution. An ε-transition in some $\widehat{\mathcal{A}}_p \in \widehat{\mathcal{S}}$ denotes an internal reduction step of the process p. Hence, every time that $\mathrm{s} \in L(q)$ and $[\![\mathrm{s} \xrightarrow{\varepsilon} \mathrm{s}']\!] \sqsubseteq \widehat{\mathcal{A}}_p$ for some $\mathrm{s} \in Q_P$ and $q \in Q^\sharp$, s' is added to $L(q)$ indicating that s' also is a possibly active point after execution of the synchronization denoted by q and before any other synchronization. This is done in (2a).

Next, consider the transitions $[\![\mathrm{E}_{l_1} \xrightarrow{l_s} \mathrm{S}_{l_s}]\!]$ and $[\![\mathrm{E}_{l_2} \xrightarrow{l_r} \mathrm{S}_{l_r}]\!]$ of two automata $\widehat{\mathcal{A}}_p$ and $\widehat{\mathcal{A}}_{p'}$. The communication is possible if E_{l_1} and E_{l_2} possibly are active at the same time, i.e. if E_{l_1} and E_{l_2} both belong to $L(q)$ for some state $q \in Q^\sharp$. If so, the transition $[\![q \xrightarrow{l_s,l_r} q_{l_s,l_r}]\!]$ is added to $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ and $L(q_{l_s,l_r})$ has to contain the active points once the synchronization has been done. Hence $L(q)$ contains the points active before the synchronization except E_{l_1} and E_{l_2}, as well as the points following the communication, namely S_{l_s} and S_{l_r}. This is done in (6). `fork`s are treated similarly in (7). A new state q_{l_f,l_f} is added to $Q^\sharp$ and $L(q_{l_f,l_f})$ is updated in the same way as for communications.

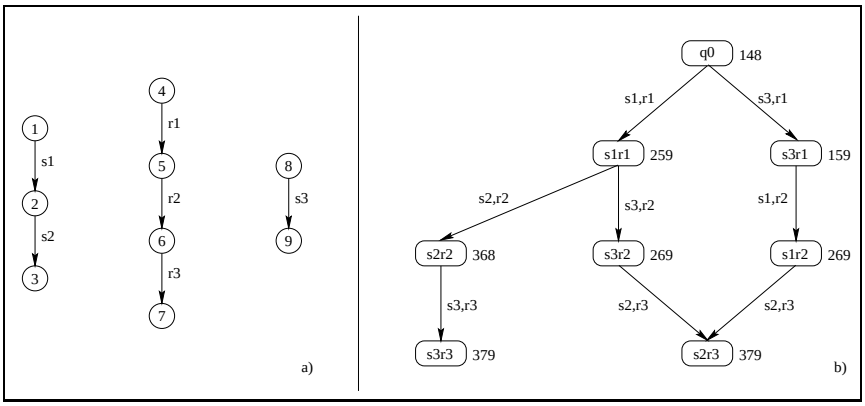


Fig. 7. a) Collection of automata for the program of Fig. 1. b) Reduced product automaton for the previous collection. Values in front of the states correspond to $L(q)$.

Figure 7 shows the reduced product automaton corresponding to the example of Figure 1. In this example, one can see that there is an exact correspondence between the states of the product automaton and the values of the function L. In this case, the reduced automaton is as precise as the product automaton and strictly described the same possible communications.

Based on $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$, we introduce a new analysis $\models^\sharp$ for a process pool P. Again, we used the sets $\widehat{\mathcal{C}}$, $\widehat{\mathcal{E}}$ defined in Section 4 as well as a reduced product automaton denoted $\widehat{\mathcal{A}}^\sharp_\otimes$. $\models^\sharp$ is obtained by substituting the reduced product automaton to the product automaton in the definition of $\models^\natural$.

Definition 8 *A triple $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes)$ defines good reduced annotations for a process pool P, denoted $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes \models^\sharp P$ iff*

(i) $\widehat{\mathcal{A}}^\sharp_\otimes = \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ for some collection $\widehat{\mathcal{S}}$ of automata such that $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}}_p)_{p \in Dom(P)}$ and $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}_p \vdash e^l$ for all $\langle p : e^l \rangle \in P$.

(ii) For all $[\![q \xrightarrow{l_s,l_r} q']\!] \sqsubseteq \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ *s.t.* $\begin{cases} (\texttt{send}\ e_0^{l_0}\ e_1^{l_1})^{l_s} \in prg \\ (\texttt{receive}\ e_2^{l_2})^{l_r} \in prg \end{cases}$, $\widehat{\mathcal{C}}(l_0) \cap \widehat{\mathcal{C}}(l_2) \neq \varnothing$ *implies* $\widehat{\mathcal{C}}(l_1) \subseteq \widehat{\mathcal{C}}(l_r)$.

Again, the existence of a least analysis in the sense of $\prec$ stems from the fact that $\{(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes) \ : \ \widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes \models^\sharp P\}$ is a Moore family.

The reduced automaton $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ collapses the states of the product automaton $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ following a given synchronization. As a consequence, any transition going out of a state $q^\natural$ in $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ corresponds to an outgoing transition going out of $q^\sharp$ in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ where $q^\sharp$ is the state approximating $q^\natural$. Hence any path in $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$ also is a path in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$. We use this observation to prove $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ may be substituted to $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}})$. Doing so, we state that the reduced automaton contains all the sequences of synchronizations of the product automaton.

Proposition 9 (Equivalence of automata) *Let P be a process pool and $\widehat{\mathcal{S}} = (\widehat{\mathcal{A}}_p)_{p \in Dom(P)}$ a collection of automata such that for all $\langle p : e^l \rangle \in P$, $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}_p \vdash e^l$. We have $\mathrm{A}^\natural_\otimes(\widehat{\mathcal{S}}) \lessdot \mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$*

Hence, $\models^\sharp$ can be used instead of $\models^\natural$ without discarding the properties established in Section 5.1. So, if $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\sharp_\otimes \models^\sharp P$ then $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}^\natural_\otimes \models^\natural P$ for some $\widehat{\mathcal{A}}^\natural_\otimes$ such that $\widehat{\mathcal{A}}^\natural_\otimes \lessdot \widehat{\mathcal{A}}^\sharp_\otimes$. We end this Section by introducing the following property about the the size of the reduced product automaton.

Proposition 10 (Size of the reduced automaton) *Let $\widehat{\mathcal{S}}$ be a collection of k automata of size $O(m)$ and let $n = km$. The reduced product automaton $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ has size $O(n^4)$.*

Proposition 10 stems from the following observations. There is at most $O(n)$ synchronization points in the program. So, the number of state in $\mathrm{A}^\sharp_\otimes(\widehat{\mathcal{S}})$ is $O(n^2)$ and the transition function has size $O(n^4)$. For a given $q \in Q^\sharp$, $L(q)$ contains all the labels of the program in the worst case, i.e. $L(q)$ has size $O(n)$. Hence L has size $O(n^3)$ and the whole size of the automaton consequently is $O(n^4)$, due to the transition function.

Because of the size limitations, we do not show how to automatically compute an analysis for a process pool P. However, one can generate a set $\mathrm{C}[P]$ of constraints such that a solution $(\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}_\otimes)$ to $\mathrm{C}[P]$ satisfies $\widehat{\mathcal{C}}, \widehat{\mathcal{E}}, \widehat{\mathcal{A}}_\otimes \models^\sharp P$. The least solution can be computed in polynomial time. One method consists of using a graph formulation of the constraints [19, 25]. Each set constrained by $\mathrm{C}[P]$ corresponds to a node in a graph G and edges are related to constraints. In this case, the complexity of the resolution of $\mathrm{C}[P]$ for a process pool P described by a program of size $O(n)$ stems from the following observations. G contains at most $O(n^4)$ vertices and $O(n^4)$ edges. A node is examined as many times as its related value is modified. Since values have size $O(n^2)$, the whole complexity of this resolution method is $O(n^6)$.

6 Application

In this Section, we comment the results given by an implementation of the CFA. We consider a virtual circuit allocation mechanism similar this of ATM [1, 26].

6.1 Virtual Circuit Allocation

Virtual circuit creation follows the scheme of Figure 8. The different nodes of the network are linked to their neighbors by control channels. These channels are used to forward a circuit creation message from the source node to the destination node. The latter creates a link with its neighbor which proceeds similarly until the source node is reached (still using control channels). Next, each intermediary node propagates the data received on its input link to its output link.

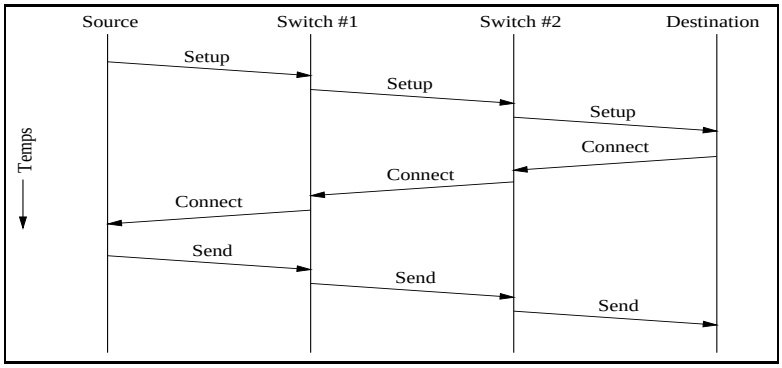
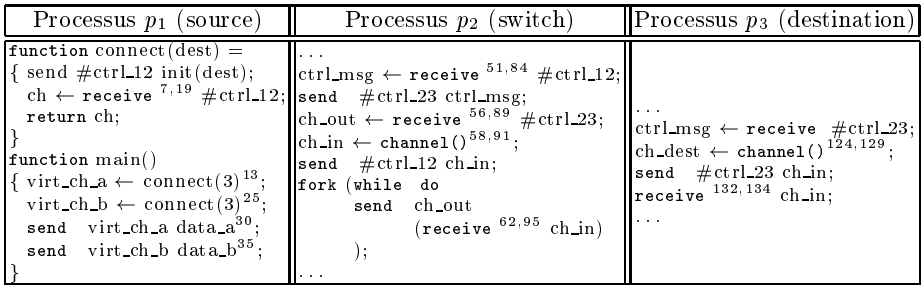


Fig. 8. Principle of virtual circuit allocation.

A full implementation of this mechanism was realized in the core language treated in this article. For simplicity, we give a simplified version written in an imperative pseudo-language (Figure 9). We only use one intermediary node.

We assume that processes p_1, p_2 and p_3 are concurrently running on three different nodes of the network. `#ctrl_12` (resp. `#ctrl_23`) is a bidirectional control channel linking the nodes of p_1 and p_2 (resp. p_2 and p_3). In addition, since p_1 creates two different circuits a and b, the pieces of code given for process p_2 are executed twice. In our implementation, we use a 0-CFA [19] for the sequential parts of the language. This enforces us to duplicate these pieces of code in order to keep enough precision. It could be avoided by using a more sophisticated analysis (1-CFA). So, in Figure 9, the two labels annotating some instructions correspond to the unique labels of two copies of the same piece of code.

p_1, p_2 and p_3 work as follows. p_1 successively asks for two new circuit creations. Function `connect` sends an initialization message on the control channel `#ctrl_12` and binds variable `virt_ch_a` to the channel name to be used for a.



Processus p_1 (source)	Processus p_2 (switch)	Processus p_3 (destination)
function connect(dest) = { send #ctrl_12 init(dest); ch ← receive ^{7,19} #ctrl_12; return ch; } function main() { virt_ch_a ← connect(3)^{13}; virt_ch_b ← connect(3)^{25}; send virt_ch_a data_a^{30}; send virt_ch_b data_b^{35}; }	... ctrl_msg ← receive ^{51,84} #ctrl_12; send #ctrl_23 ctrl_msg; ch_out ← receive ^{56,89} #ctrl_23; ch_in ← channel()^{58,91}; send #ctrl_12 ch_in; fork (while do send ch_out (receive ^{62,95} ch_in)); ...	... ctrl_msg ← receive #ctrl_23; ch_dest ← channel()^{124,129}; send #ctrl_23 ch_in; receive ^{132,134} ch_in; ...

Fig. 9. Implementation of the virtual circuit allocation mecanism. Procedure connect as well as the pieces of code of processes p_2 and p_3 are executed twice.

This scheme is repeated for b. Finally, data are transmitted on `virt_ch_a` and `virt_ch_b`.

p_2 corresponds to the node between the source and the destination. When a virtual circuit creation request is received on `#ctrl_12`, it is transmitted to the destination on the control channel `#ctrl_23`. p_2 next receives the output channel name to use and binds variable `ch_out`. Next, it creates a new channel `ch_in` which is sent to the source process. The last operation consists of creating a new process which forwards on `ch_out` the values received on `ch_in`. The whole process is repeated for the virtual circuit b.

Finally, p_3 corresponds to the destination node. When the control message is received, p_3 creates a new channel and binds variable `ch_in`. This channel name is transmitted to p_2 via control channel `#ctrl_23`. Once the circuits are created, p_3 receives data on a and b.

6.2 Analysis

We used our analysis to check the correctness of a full implementation of the program described in Figure 9. Here we present the results we obtained, translated to the program of Figure 9. Figure 10 gives the abstract values obtained for the points of interest and for relevant variables. As indicated above, some pieces of code are executed twice. In Figure 10, we indicate for variables which execution is referenced by (a) and (b).

The main observation concerns the abstract values received by p_3 at points 132 and 134. These values, $\{30\}$ and $\{35\}$, are the ones emitted by p_1 on circuits a and b. This validates the mechanism. The values received on a (resp. b) are the ones emitted on this channel and only these ones. This guarantees that p_2 does not invert the data received on its input links and forwarded on its output links.

Concerning the virtual circuit a, one can check that the variables `virt_ch_a` and `ch_dest` (a) corresponding to the extremities of the circuit are bound to $\{58\}$ and $\{124\}$ which correspond to the channel names created at these points. The new process created by p_2 forwards on a name created at point 124 the data

Label	7	13	19	25	30	35	51	56	58
Value	{58}	{13}	{91}	{25}	{30}	{35}	{13}	{124}	{58}

Label	62	84	89	91	95	124	129	132	134
Value	{30}	{25}	{129}	{91}	{35}	{124}	{129}	{30}	{35}

Variable	#ctrl_12	#ctrl_23	virt_ch_a	virt_ch_b	ctrl_msg (a)	ctrl_msg (b)
Value	{1}	{2}	{58}	{91}	{13}	{25}

Variable	ch_in (a)	ch_in (b)	ch_out (a)	ch_out (b)	ch_dest (a)	ch_dest (b)
Value	{58}	{91}	{124}	{129}	{124}	{129}

Fig. 10. Abstract values attached to the labels of the program of Figure 9.

received on a name created at point 58 (variables `ch_in` (a) and `ch_out` (a)). Thus, the transmission of data from the source to the destination is ensured. In addition, it is possible to check that the channel names of abstract value {58} and {124} are not used elsewhere in the program. So, the data emitted on the virtual circuit a only are transmitted to the destination node. Similar observations can be done for the virtual circuit b.

Finally, let us take note that in order to obtain these results, the analysis has to use an approximation of the communication topology of the program. The same control channels `#ctrl_12` et `#ctrl_23` are used for both circuit creation. An analysis based on Equation (1) could not obtain these results. The result would be `virt_ch_a` = `virt_ch_a` = $\{58, 91\}$ which does not enable us to check that no confusion is done between the data transmitted on both circuits.

We also used this analysis in order to verify security properties for an auction distributed application. This system is made of a server and several clients who send to the server public data (e.g. their price) as well as confidential data (e.g. their credit card number). Each client is linked to the server by one communication channel. The server broadcasts the public data and conserves the private ones. We show that, in our implementation, the only public data are actually broadcasted.

7 Conclusion

In this article, we introduced a static analysis able to depict a conservative image of the communication topology of a concurrent system written in Concurrent ML. The dynamic aspects of concurrency are managed, including `fork`s, the transmission of functions and channel creations. This analysis is a CFA which builds a finite automaton in order to increase its precision. It was implemented and results are discussed in Sections 2 and 6.

Using product automata allows us to derive a fine approximation of the topology, as shown in Section 2. This represents the main contribution of this article. It enables to minimize the abstract value attached to reception points. Gains are twofold. First, this increase the precision of the analysis for the sequential part of the program following the reception. Second, since channel names are

potentially communicated, we also define more concisely the pairs of possible emitters and receptors for further communications.

We believe that the gains due to the introduction of topological information in the CFA improve the analysis of many kinds of applications. For quite common deterministic communication schemes, such as a sequence of communications on the same channel between two processes, this makes possible to distinguish between the values sent at each stage. For more complicated communication schemes, precision may decrease at some points in the program, for instance because of alternatives, loops or non-determinism. However, precision will increase again once the ambiguities disappear, for instance after a global synchronization.

Now, we are interested in mixing it with other analyses, in order to increase the conciseness of the annotations. We focus on a binding time analysis [8, 12] which uses topological informations during its analysis of programs. In this context, our CFA would enable to distinguish between the static and dynamic values sent on the same channel, while usual analyses consider a channel dynamic as soon as a dynamic value is sent on it.

References

1. Antony Alles. ATM Internetworking. Technical report, 1995. CISCO Systems Inc.
2. T Amtoft, Flemming Nielson, and Hanne Riis Nielson. Behaviour analysis and safety conditions: a case study in CML. In *FASE'98*, number 1382 in Lecture Notes in Computer Science, pages 255–269. Springer-Verlag, 1998.
3. Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'92*. ACM, 1992.
4. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *Concur'98*, number 1466 in Lecture Notes in Computer Science, pages 84–98. Springer-Verlag, 1998.
5. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis of processes for no read-up and no write-down. In *FOSSAC'99*, number 1578 in Lecture Notes in Computer Science, pages 120–134. Springer-Verlag, 1999.
6. Christopher Colby. Analyzing the communication topology of concurrent programs. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'95*, pages 202–213. ACM, 1995.
7. Charles Consel and Olivier Danvy. Partial evaluation: Principles and perspectives. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'93*. ACM, 1993.
8. Marc Gengler and Matthieu Martel. Self-applicable partial evaluation for the pi-calculus. In *Partial Evaluation and Semantics-Based Program Manipulation, PEPM'97*, pages 36–46, 1997.
9. Marc Gengler and Matthieu Martel. Des étages en Concurrent ML. In *Rencontres Francophones du Parallélisme, Renpar10*, 1998.
10. Suresh Jagannathan. Locality abstractions for parallel and distributed computing. In *International Conference on Theory and Practice of Parallel Programming*, number 907 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

11. Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communications for asynchronous concurrent programming languages. In *SAS'95*, volume 983 of *LNCS*, pages 225–242. Springer-Verlag, 1995.
12. Mihnea Marinescu and Benjamin Goldberg. Partial evaluation techniques for concurrent programs. In *ACM-SIGPLAN Symposium on Partial Evaluation and Semantic Based Program Manipulations PEPM'97*, pages 47–62. ACM, 1997.
13. Nicolas Mercouroff. An algorithm for analyzing communicating processes. *Lecture Notes in Computer Science*, 598:312–325, 1992.
14. Flemming Nielson and Hanne Riis Nielson. Constraints for polymorphics behaviours of Concurrent ML. In *Constraints in Computational Logics*, number 845 in Lecture Notes in computer Science, pages 73–88. Springer-Verlag, 1994.
15. Flemming Nielson and Hanne Riis Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'94*, pages 84–97. ACM, 1994.
16. Flemming Nielson and Hanne Riis Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In *TAPSOFT'95*, number 915 in Lecture Notes in Computer Science, pages 590–604. Springer-Verlag, 1995.
17. Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *ACM-SIGPLAN Symposium on Principles of Programming Languages POPL'97*, pages 332–345. ACM, 1997.
18. Flemming Nielson and Hanne Riis Nielson. Communication analysis for Concurrent ML. In *ML with Concurrency*, Monograph in Computer Science, pages 185–251. Springer, 1999.
19. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.
20. John H. Reppy. An operational semantics of first-class synchronous operations. Technical Report TR-91-1232, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1991.
21. John H. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1992.
22. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.
23. Olin Shivers. Control flow analysis in scheme. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, PLDI'88*, pages 164–174. ACM, 1988.
24. Olin Shivers. *Control Flow Analysis of Higher Order Languages*. PhD thesis, Carnegie Mellon University, 1991. Technical Report CMU-CS-91-145.
25. Kirsten L Solberg, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for CML. In *ACM-SIGPLAN International Conference on Functional Programming, ICFP'97*, pages 38–51. ACM, 1997.
26. Andrew S. Tanenbaum. *Computer Networks, Third Edition*. Prentice Hall, 1996.