Distributing Automata for Asynchronous Networks of Processors

Benoit Caillaud, Paul Caspi, Alain Girault, Claude Jard

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Distributing Automata for Asynchronous Networks of Processors

Benoît Caillaud, Paul Caspi, Alain Girault and Claude Jard

N° 2341

Septembre 1994

PROGRAMMES 1 et 2

Architectures parallèles, bases de données, réseaux et systèmes distribués

Calcul symbolique, programmation et génie logiciel

ISSN 0249-6399

Distributing Automata for Asynchronous Networks of Processors

Benoît Caillaud*, Paul Caspi**, Alain Girault*** and Claude Jard****

Programmes 1 et 2 — Architectures parallèles, bases de données, réseaux
et systèmes distribués — Calcul symbolique, programmation et génie logiciel
Projets Pampa et Spectre

Rapport de recherche n° 2341 — Septembre 1994 — 16 pages

Abstract: This paper addresses the problem of distributed program synthesis. In the first part, we formalize the distribution process and prove its correctness, i.e. that the initial centralized program's behavior is equivalent to the corresponding distributed's one. In order to achieve that, we first represent the program by a finite transition system, labeled by the program's actions. Then we derive an independence relation over the actions from the control and data dependencies. This leads to represent the program by an order-automaton, whose transitions are labeled partial orders coding for an action and its dependencies with other actions. In the second part, we show how such an order-automaton can be practically used to derive a distributed program.

Key-words: theory of parallel and distributed computation, automata and formal languages

(Résumé : tsvp)

This work has been partially supported by *PRC* C^3 (CNRS), *GRECO Automatique action C2A* and *Ministère de l'Enseignement Supérieur et de la Recherche*

*LFCS, Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Scotland UK,
Email: bc@dcs.ed.ac.uk
**VERIMAG U.M.R. C9939, Miniparc - ZIRST, 38041 Montbonnot Saint Martin, FRANCE,
Email: Paul.Caspi@imag.fr, Tel: (+33) 76 90 96 33, Fax: (+33) 76 41 36 20
VERIMAG is a joint laboratory of CNRS, Institut National Polytechnique de Grenoble, Université J. Fourier and VERILOG S.A. associated with IMAG.
***SCHNEIDER ELECTRIC Département PLI/SES, usine M3, 38050 Grenoble cedex 09, FRANCE,
Email: Alain.Girault@imag.fr, Tel: (+33) 76 90 96 33, Fax: (+33) 76 41 36 20
****IRISA, Campus de Beaulieu, 35042 Rennes cedex, FRANCE,
Email: Claude.Jard@irisa.fr, Tel: (+33) 99 84 71 93, Fax: (+33) 99 38 38 32

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

Répartition d'Automates pour un Réseau Asynchrone de Processeurs

Résumé : Cet article porte sur le problème de la construction des programmes répartis. Dans la première partie, nous formalisons le processus de répartition et nous prouvons sa correction, c'est-à-dire que le comportement du programme centralisé initial est équivalent à celui du programme réparti correspondant. Dans ce but, nous représentons le programme par un système de transitions étiquetées par les actions du programme. Puis nous établissons une relation de commutation sur les actions en fonction des dépendances de contrôle et de données. Ceci nous amène a représenter le programme sous la forme d'un automate d'ordres, dont les transitions sont étiquetées par des ordres partiels exprimant les liens de dépendence entre une actions et les autres actions du programme. Dans la seconde partie, nous montrons comment un tel automate d'ordres peut être utilisé en pratique pour obtenir un programme réparti.

Mots-clé : théorie de la répartition, automates et langages formels

1 Introduction

We consider the problem of **distributed program synthesis**, which becomes of paramount importance for the design of programs on distributed computers or networks of processors.

In distributed machines, each process has its own memory and processes must communicate explicitly by sending and receiving messages. As a result, the programmer faces the enormously difficult task of managing the entire parallel execution: he has to design the different codes and data to be distributed on the processors, and to manage communication among tasks. This, in addition to being error-prone and time-consuming, generally leads to non-portable code. Hence, parallelizing compilers for distributed machines have been an active area of research recently.

Research has benefited from progress in compiler technology which permits to forecast different and **large application areas**:

- Automated distribution of sequential programs: the programming model, imperative sequential programming (e.g. Fortran) is left unchanged, but a compiler is required to express its work in terms of a distributed execution model. The goal is to obtain better performances. Currently, the main parallelization technique in this area is the "data-driven" approach, which consists in automatically creating communicating processes from sequential code plus data decomposition specifications. The most usual code generation is based on the *Single Program Multiple Data* principle [6] and on the standard notion of "refresh" of remote variables [1]. The correctness of the distribution is achieved when, given corresponding initial data, either the source sequential code and the parallel generated one produce corresponding results, or both of them diverge [3].

- Automated distribution of reactive systems: in that case, increasing performances is not the only motivation, distribution can be driven by the location of sensors and actuators. Synchronous languages are good candidates to program reactive systems [5, 4]. They demand a deterministic style of programming, which simplifies the development process. But, in contrast with the distribution of sequential imperative programs, reactivity must be carefully preserved by the distributed implementation [7].

- Protocol synthesis: the starting point is the service specification. The distributed protocol is then synthesized by applying general transformations which preserve the original semantics. This domain is the most difficult to automatize, due to the usual non-determinism of protocol service specifications [12, 16].

All these application areas, listed by increasing complexity, require a formal treatment in order to be able to define transformations rigorously and to prove their properties. We are involved in the **formal characterization of the distributed behaviors obtained by applying these transformations**. The major goal of this paper is to give a first contribution to a sound theoretical foundation.

Formal works on automated distribution are still young. For this reason, they are all based on simple models like automata or nets [11]. We chose to dig this way by using finite transition systems, which appeared well suited to capture the control structures of the programs we consider. Consequently, we focus on the problem of formalizing the distribution process and on the characterization of the generated distributed behaviors. Starting from a single automaton, we decide to create parallelism by using some given independence relation between the elements of the alphabet. This knowledge is usually deduced from a static analysis of the source program, identifying the control and data dependencies. Since independence is generally not symmetric (e.g. considering read and write operations on variables), we use the generalization of the well-known theory of trace-languages [14, 9] to semi-commutations [8].

The independence relation identifies the different admissible re-orderings of transitions that can occur in the distributed implementation. This will be formally captured by the notion of word and language closure. The main result of this paper is to show how the distribution process is able to generate a code whose behaviors are as close as possible to the set of admissible behaviors. We show the consequences of message synchronization and the difficulty of non-deterministic choices. To do that, we use an original intermediate representation: order-automata.

Apart from the applications mentioned in the beginning, there are some related works on models. As far as we know, the closest contribution comes from A. Petit [15].

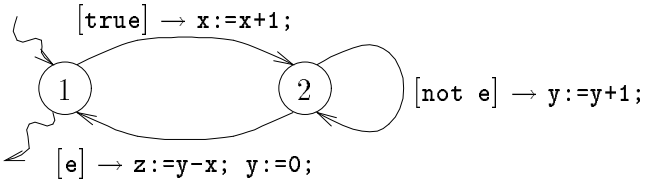
In this related work a new characterization of recognizable trace languages is given: a trace language is recognizable if and only if it is recognized by a so-called "*distributed automaton*". The key point of this work is that distributed automata can easily be mapped onto parallel architectures[1]. However these automata hardly suit distributed architectures because of the existence of a process in charge of every processor synchronization. Moreover the case of rational trace languages cannot be addressed with this method. Finally "*distributed automata*" are tightly coupled to trace theory. Its generalization to non-symmetric independence relations seems very unlikely.

The paper is organized as follows.

We first present a theoretical model based on labeled partial orders. This model allows us to code with a labeled partial order the closure of words modulo an independence relation. Then we propose two algorithms generating order-automata. To some extent, these order-automata recognize closure of regular languages modulo an independence relation. The last part of the paper shows that our formal model can be applied to practical distribution applications.

2 Labeled Partial Orders

Partial order theory helps in formalizing automata distribution. This section deals with some basic tools on partial orders. Usual definitions and concepts of partial order theory are used whenever possible [10]. All along this section, concepts and properties will be illustrated by small examples. These examples are based on a synchronous reactive program [5, 4], represented by the following automaton with guarded commands on the transitions:

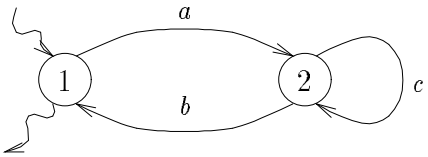


This program is controlled by the environment through event e. At each trigger, e can be either true or false. The transitions are labeled a, b and c, according to the following table:

action	label
[true] → x:=x+1;	a
[e] → z:=y-x; y:=0;	b
[not e] → y:=y+1;	c

From now on, we consider the corresponding automaton P over the alphabet $\Sigma = \{a, b, c\}$:

[1] Such as shared memory multiprocessors.



State 1 is the initial state and only accepting state of P, denoted by small curved arrows.

The set of states of automaton P is denoted Q_P, $q_P \in Q_P$ denotes the initial state of P, $F_P \subset Q_P$ denotes the set of accepting (or final) states, Σ_P is the action alphabet of P and $\longrightarrow_P \subset Q_P \times \Sigma_P \times Q_P$ denotes its transition relation. For the sake of simplicity, only deterministic automata will be considered.

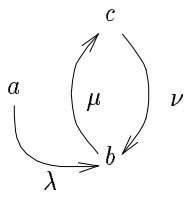
The distribution of automaton P relies on an independence (or commutation) relation: this relation $\propto$ is derived from the data and control dependencies in the program. For each action, we define the sets USE and DEF of variables respectively used and assigned by this action. For the example, we have:

action	USE	DEF
a	$\{x\}$	$\{x\}$
b	$\{x, y\}$	$\{y, z\}$
c	$\{y\}$	$\{y\}$

The independence relation is defined by the following rule:

$$\forall \alpha, \beta \in \Sigma,\ \alpha \propto \beta \iff DEF(\alpha) \cap USE(\beta) = \emptyset \text{ and } \alpha \neq \beta$$

Thus, $\propto = \{(a, c), (c, a), (b, a)\}$, which is clearly not symmetric. Hence the dependence relation $\not\propto = \Sigma^2 \setminus \propto$ is:



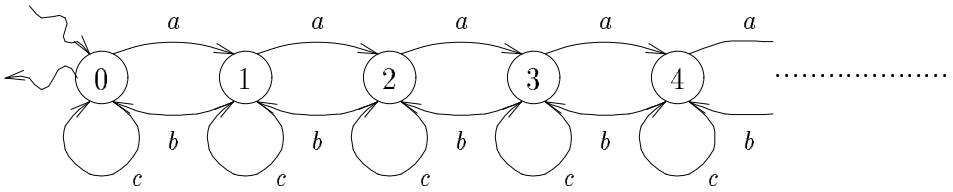
For the sake of clarity the edges of the dependence graph are labeled by greek letters: λ, μ, ν.

The rewriting relation (or semi-commutation [8]) $\rightharpoonup$ on $\Sigma^\star$ generated by $\propto$, is the transitive and reflexive closure of the relation $\rightharpoonup_1$. Relation $\rightharpoonup_1$ is the least relation verifying: $\forall u, v \in \Sigma^\star, \forall a, b \in \Sigma, u \cdot a \cdot b \cdot v \rightharpoonup_1 u \cdot b \cdot a \cdot v \iff (a, b) \in \propto$. For any pair of words $u, v \in \Sigma^\star$, $u \rightharpoonup v$ means that v is an implementation of u, according to the commutation relation $\propto$. Notice that relation $\rightharpoonup$ is not symmetric in general: this is the very reason why it is called semi-commutation.

The right-closure by $\rightharpoonup$ of a word $u \in \Sigma^\star$ and of a language $A \subset \Sigma^\star$ are denoted $[u]_\rightharpoonup = \{u | u \rightharpoonup v\}$ and $[A]_\rightharpoonup = \bigcup_{u \in A} [u]_\rightharpoonup$.

The length of a word $u \in \Sigma^\star$ is denoted $|u|$, and for any $i = 1 \ldots |u|$, u_i denotes the i^{th} letter of u.

The language of automaton P, denoted $||P||$, is $(ac^\star b)^\star$. The right-closure of $||P||$ by the rewriting relation $\rightharpoonup$ is accepted by the following transition system:



This transition system is minimal, therefore the right-closure of the language of P is not regular.

This example shows that it is, in general, not possible to code the right-closure of the language of an automaton by a finite transition system on words. This is the very reason why a formalism based on partial orders is introduced.

A labeled partial order (l.p.o. for short) is the triple $\Theta = (E_\Theta, \leq_\Theta, \varphi_\Theta)$, where $\leq_\Theta$ is a partial order relation over E_Θ and $\varphi_\Theta : E_\Theta \to \Sigma$ a labeling of elements of E_Θ. A l.p.o. is non-autoconcurrent if and only if every pair of elements labeled by the same letter are comparable.

The projection of a l.p.o. Θ on $A \subset \Sigma$ is the sub-order consisting in all elements of E_Θ labeled by letters in A: $\Theta|_A = \left(\varphi_\Theta^{-1}(A), \leq_\Theta \cap \left(\varphi_\Theta^{-1}(A) \times \varphi_\Theta^{-1}(A)\right), \varphi_\Theta|_{\varphi_\Theta^{-1}(A)}\right)$.

An ideal (resp. filter) of a partial order $\leq$ on E, is a subset $I \subset E$ (resp. $F \subset E$) closed by precedence (resp. closed by precedence of the reverse order): $\forall i \in I, e \in E,\ e \leq i \Rightarrow e \in I$ and $\forall f \in F, e \in F,\ f \leq e \Rightarrow e \in F$.

For any word $u \in \Sigma^\star$, $\vec{u}$ denotes the l.p.o. $(\{1, \ldots, |u|\}, \leq_N, i \mapsto u_i)$. Informally, $\vec{u}$ is a totally ordered set labeled by the successive letters of u.

For any mapping $f : A \to B$ and any binary relation $\rho \subset B^2$, relation $\Re_{\rho,f} \subset A^2$ is defined as follows: $\forall a, b \in A,\ (a,b) \in \Re_{\rho,f} \iff (f(a), f(b)) \in \rho$.

The linear (total orders) extensions language of a finite l.p.o. Θ is the language $\mathcal{L}(\Theta) \subset \Sigma^\star$ such that: $\forall u \in \Sigma^\star,\ u \in \mathcal{L}(\Theta) \iff \vec{u}$ is isomorphic to a linear extension of Θ. More generally, words and labeled total orders will be used indifferently. For any set of labeled partial orders A, $\mathcal{L}(A) = \bigcup_{\Theta \in A} \mathcal{L}(\Theta)$ denotes the set of all linear extensions of orders of A.

The following lemma justifies the use of partial orders. It states that the right-closure of a word by the rewriting relation $\rightharpoonup$ is the set of linear extensions of a l.p.o.:

Lemma 1 *Let $u \in \Sigma^\star$ and $\propto \subset \Sigma^2$. The rewriting on $\Sigma^\star$ generated by $\propto$ is denoted $\rightharpoonup$.*

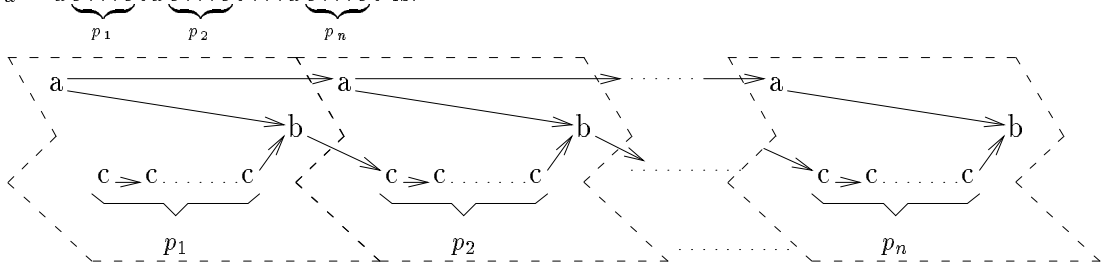
$$[u]_{\rightharpoonup} = \mathcal{L}\left(\Gamma_{u,\not\propto}\right)$$

Where $\Gamma_{u,\not\propto} = \left(E_{\vec{u}}, \left(\leq_{\vec{u}} \cap \Re_{\not\propto,\varphi_{\vec{u}}}\right)^\star, \varphi_{\vec{u}}\right)$.

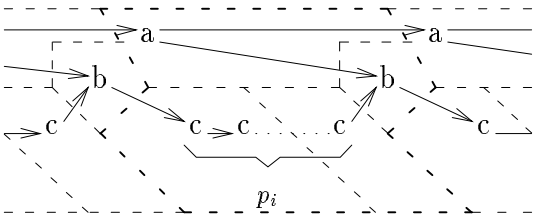
This lemma extends a result given in [9, 17] for the particular case of a symmetric independence relation.

Notice that the l.p.o. coding for the right-closure of a word modulo an irreflexive independence relation is non-autoconcurrent. From now on the independence relation is assumed to be irreflexive.

The covering graph of such a labeled partial order is called dependence graph. For instance, the dependence graph of the order coding for the right-closure by $\rightharpoonup$ of the word $u = a\underbrace{c\ldots c}_{p_1}ba\underbrace{c\ldots c}_{p_2}b\ldots a\underbrace{c\ldots c}_{p_n}b$ is:



Informally, this graph is the "*concatenation*" of tiles. Moreover, each tile can be decomposed in smaller patterns:



In order to achieve this tile composition, a concatenation on orders is defined:

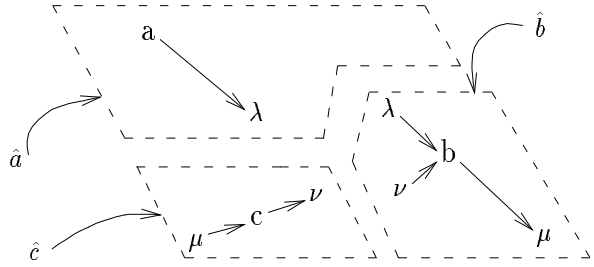
Definition 1 *Let Θ and Φ be two non-autoconcurrent l.p.o. The concatenation of Θ and Φ, denoted $\Theta \cdot \Phi$ is the least non-autoconcurrent l.p.o. Λ such that $E_\Lambda = E_\Theta \oplus E_\Phi$, $\leq_\Lambda \supset \leq_\Theta \oplus \leq_\Phi$ and $\varphi_\Lambda = \varphi_\Theta \oplus \varphi_\Phi$, admitting E_Θ as an ideal and E_Φ as a filter*[2].

In other words, the concatenation of Θ and Φ is such that any element of Φ labeled with a letter a is greater than any occurrence of the letter a in Θ.

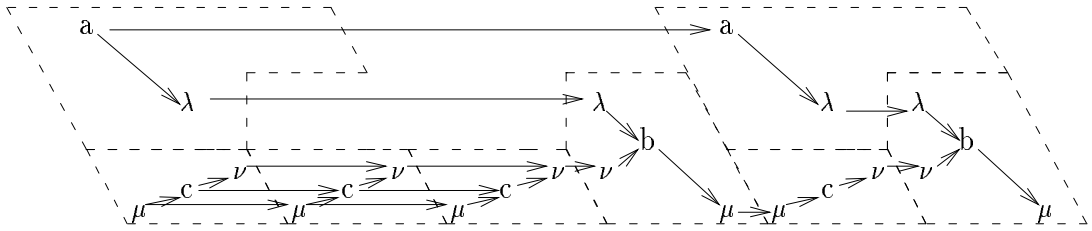
The set of non-autoconcurrent l.p.o. over alphabet Σ, with the order concatenation $\cdot$ is a monoïd. The neutral element is the empty l.p.o.: $\vec{\epsilon} = (\emptyset, \emptyset, \emptyset)$. Associativity of the concatenation is easily checked.

An order-automaton is an automaton with transitions labeled by labeled partial orders. The language $||Q||$ of an order-automaton Q is the set of concatenations (as defined above) of labels of all accepting sequences — in its usual meaning.

The concatenation of orders and a synchronization alphabet (λ, μ, ν in the example) are used jointly in order to compose the tiles given below:

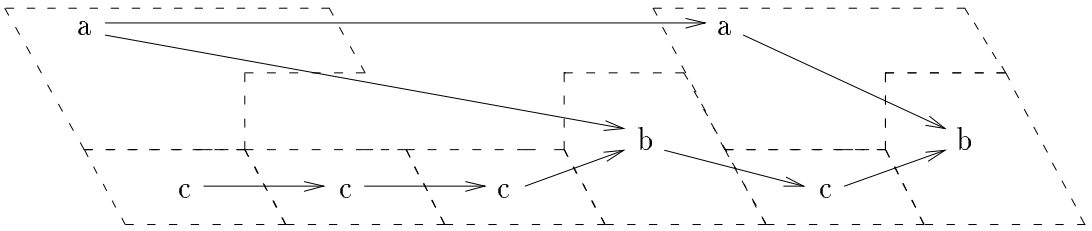


For instance the word $v = ccacbcab$ is a rewriting of the word $u = acccbacb \in ||P||$ with the relation $\rightharpoonup$. The l.p.o. $\hat{u} = \hat{a} \cdot \hat{c} \cdot \hat{c} \cdot \hat{c} \cdot \hat{b} \cdot \hat{a} \cdot \hat{c} \cdot \hat{b} \cdot$ is:

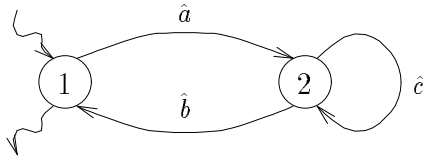


The word v is a linear extension of the projection of $\hat{u}$ on Σ:

[2] $\oplus$ being the exclusive union between sets



Finally, it should be noticed that the closure of $||P||$ by the relation $\rightharpoonup$ is the set of all linear extensions of the projection on Σ of the language of the order-automaton $\hat{P}$:



More formally: $[\![|P|]\!]_{\rightharpoonup} = \mathcal{L}\left(\left\|\hat{P}\right\|_{\Sigma}\right)$.

3 Synthesis of Order-Automata

In the previous section, order-automata have been introduced. With the help of a small example, it has been stated that the right-closure of a regular language modulo the rewriting relation generated by an independence relation could be coded by an order-automaton.

This section is entirely devoted to the synthesis of such order-automata. The first part of this section deals with the basic synthesis of order-automata. The second part deals with the synthesis of optimized order-automata. The third and last part address the correctness issues of these transformations.

In the first place and for the sake of clarity, a few notations must be introduced. For any binary relations ψ, φ over alphabet Σ, and any letter $a \in \Sigma$, the following three relations are defined: $\varphi a = \varphi \cap (\Sigma \times \{a\})$ denotes the couples of φ ending with the letter a. The set of couples of φ starting with the letter a is denoted $a\varphi = \varphi \cap (\{a\} \times \Sigma)$. The transitive one-step composition of the relations φ and ψ is: $\varphi \cdot \psi = \{(u,v) | \exists w \in \Sigma, (u,w) \in \varphi, (w,v) \in \psi\}$.

3.1 Basic Order-Automata

A synchronization alphabet enables to preserve dependencies between letters of Σ, according to the dependence relation $\not\propto = \Sigma^2 \setminus \propto$. This alphabet is merely the dependence relation without the self-dependence couples: $\rho = \not\propto \setminus \{(a,a)\}_{a \in \Sigma}$. Therefore the labels of the transitions of the order-automaton are labeled partial orders over the alphabet $\Sigma \oplus \rho$.

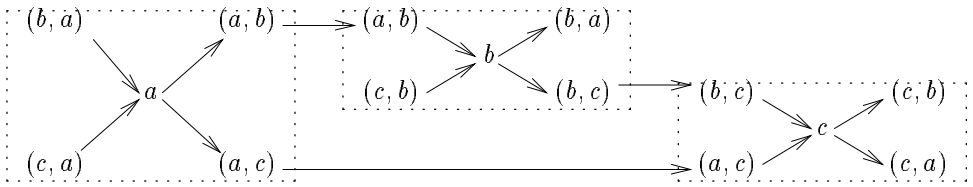
The order-automaton $\mathcal{F}(P)$ is defined by replacing each transition $q \xrightarrow{a} q'$ of automaton P by the transition $q \xrightarrow{\mathcal{F}(a)} q'$, where the label of the transition is:

$$\mathcal{F}(a) = \begin{pmatrix} \forall \xi \in \rho a & & & \forall \chi \in a\rho \\ \xi & & & \chi \\ \vdots & \searrow & \nearrow & \vdots \\ \vdots & \vdots \; a & \vdots & \vdots \end{pmatrix}$$

Accepting states and the initial state are unchanged.

The order-automaton $\hat{P}$ given at the end of section 2 is produced by this algorithm.

Unfortunately the algorithm produces useless synchronizations. For instance, the image of the word abc through morphism[3] $\mathcal{F}$ with an empty independence relation is the l.p.o. $\mathcal{F}(a) \cdot \mathcal{F}(b) \cdot \mathcal{F}(c)$:



the synchronization $a \longrightarrow (a,c) \longrightarrow (a,c) \longrightarrow c$ is useless since the path $a \longrightarrow (a,b) \longrightarrow (a,b) \longrightarrow b \longrightarrow (b,c) \longrightarrow (b,c) \longrightarrow c$ already ensures the comparability of the occurrences of letters a and c.

3.2 Optimized Order-Automata

The aim of the optimization is to reduce the number of synchronizations as much as possible. More precisely a synchronization will be generated only when required, that is when the synchronization ensures a new comparability between the occurrences of two letters in Σ.

For this purpose, some information on the "*past*" of a computation that has left the order-automaton in a given state must be coded in this state. That information, called *context*, is the set of synchronizations that might occur in a transition starting from this state.

Order-automaton $\mathcal{F}'(P)$ is defined as follows:

$$\begin{cases} Q_{\mathcal{F}'(P)} & = & Q_P \times 2^\rho \\ q_{\mathcal{F}'(P)} & = & (q_P, \emptyset) \\ F_{\mathcal{F}'(P)} & = & F_P \times 2^\rho \\ \longrightarrow_{\mathcal{F}'(P)} & = & \bigcup_{p \in 2^\rho} {\mathcal{F}'}_p(\longrightarrow_P) \end{cases}$$

Transition labels of the order-automaton are labeled partial orders over alphabet $\Sigma \oplus \rho$. For any context p and transition (q, a, q') a transition of the order-automaton is generated:

$$ {\mathcal{F}'}_p(q,a,q') = ((q,p), {\mathcal{F}'}_p(a), (q, \mathcal{C}_a(p))) $$

The new context is: $\mathcal{C}_a(p) = (p - \rho a - (pa \cdot a\rho)) \cup a\rho$. Notice that synchronizations in p are retained in the new context if and only if they are not covered by new synchronizations. The label of the transition is:

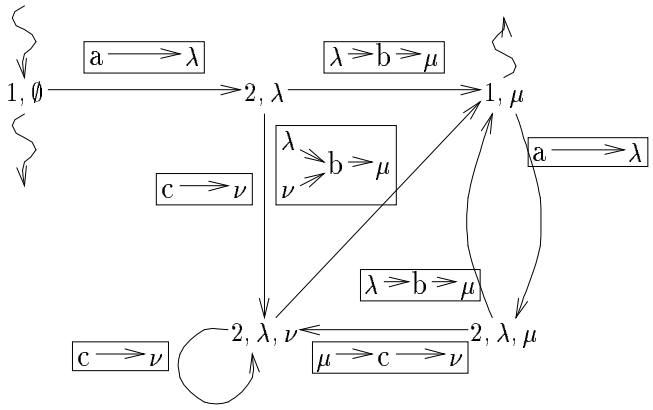
$$ {\mathcal{F}'}_p(a) = \begin{pmatrix} \forall \xi \in pa & & & & \forall \chi \in a\rho \\ \xi & & & & \chi \\ \vdots & & \searrow & \nearrow & \vdots \\ \vdots & & \vdots \; a \; \vdots & & \vdots \end{pmatrix} $$

Only reachable states should be considered. They are in finite number since alphabet Σ is finite. The algorithm for generating all of them relies on a traversal of the transition graph.

Informally the optimization of the distribution of behavior $u \in \Sigma^\star$ is roughly the computation of the covering graph of the order $\mathcal{F}(u)|_\Sigma$ where each edge denotes a synchronization.

The order-automaton given below is the optimized distribution of automaton P and independence relation $\propto$ as given in section 2:

[3]Function $\mathcal{F}$ induces a monoïd morphism from $\Sigma^\star$ to the set of labeled partial orders over $\Sigma \oplus \rho$. This morphism is also denoted $\mathcal{F}$.



3.3 Correctness

Beforehand, we generalize $\mathcal{F'}_p$ to words. For all $p \in 2^\rho$, its inductive definition is: $\mathcal{F'}_p(\epsilon) = \vec{\epsilon}$ and $\forall a \in \Sigma, \forall u \in \Sigma^\star, \mathcal{F'}_p(au) = \mathcal{F'}_p(a) \cdot \mathcal{F'}_{\mathcal{C}_a(p)}(u)$. When the context is omitted, the empty one is assumed: $\mathcal{F'}(u) = \mathcal{F'}_\emptyset(u)$.

Mapping $\mathcal{C}$ is also extended: $\forall p \in 2^\rho, a \in \Sigma, u \in \Sigma^\star,\ \mathcal{C}_{au}(p) = (\mathcal{C}_u \circ \mathcal{C}_a)(p)$. And $\mathcal{C}_\epsilon(p) = p$.

In this section function f denotes either $\mathcal{F}$ or $\mathcal{F'}$.

Lemma 2 $f(\|P\|) = \|f(P)\|$.

proof It is sufficient to prove that f maps onto $\|f(P)\|$ and is surjective.

- Let $u \in \|P\|$. There exists an accepting sequence : $q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} q_2 \ldots \xrightarrow{u_n} q_n$ where q_0 is the initial state and q_n is an accepting state. Since the automaton P is deterministic, this sequence is unique.

  A unique sequence of the order-automaton is associated with this sequence:

  $$q'_0 \xrightarrow{\hat{u}_1} q'_1 \xrightarrow{\hat{u}_2} q'_2 \ldots \xrightarrow{\hat{u}_n} q'_n$$

  More precisely:

  - if $f = \mathcal{F}$ then $q'_i = q_i$ and $\hat{u}_i = \mathcal{F}(u_i)$.
  - if $f = \mathcal{F'}$ then $q'_i = (q_i, c_i)$ where $(c_i)_{i=1\ldots n}$ is a sequence of contexts, $c_0 = \emptyset$ and $\forall i = 1 \ldots n,\ \hat{u}_i = \mathcal{F'}_{c_{i-1}}(u_i),\ c_i = \mathcal{C}_{u_i}(c_{i-1})$.

  The existence of this sequence implies that $f(P)$ accepts the order $f(u)$.

- For any order $\Theta \in \|f(P)\|$, there exists an accepting sequence: $q_0 \xrightarrow{o_1} q_1 \xrightarrow{o_2} q_2 \ldots \xrightarrow{o_n} q_n$. Where q_0 is the initial state of $f(P)$, q_n is an accepting state of $f(P)$ and $\Theta = o_1 \cdot \ldots \cdot o_n$. This sequence can be projected on alphabet Σ. Also, to each state of the sequence corresponds a unique state of the automaton P. Since each order o_i contains exactly one occurrence of a letter of Σ, we obtain the following sequence: $q'_0 \xrightarrow{u_1} q'_1 \xrightarrow{u_2} q'_2 \ldots \xrightarrow{u_n} q'_n$ with $\forall i = 1 \ldots n,\ u_i = o_i|_\Sigma$. States of the new sequence are:

  - if $f = \mathcal{F}$ then $q'_i = q_i$.
  - if $f = \mathcal{F'}$ then $q_i = (q'_i, c_i)$.

Actually this sequence is the accepting sequence of the word u by the automaton P.

Lemma 3 *Let $u \in \Sigma^\star$.*

$$f(u)|_\Sigma \;=\; \Gamma_{u,\not<}$$

proof This lemma will be proved by induction on the size of the word u. Let $u \in \Sigma^\star$ with $|u| \leq k$, $k \geq 0$. For all $a \in \Sigma$.

- If $f = \mathcal{F}$ then the following holds by definition of $\mathcal{F}$: $\mathcal{F}(u \cdot a)|_\Sigma \;=\; (\mathcal{F}(u) \cdot \mathcal{F}(a))\,|_\Sigma$.
  Concatenation of $\mathcal{F}(u)$ and $\mathcal{F}(a)$ establishes comparabilities between the occurrence of letter a in $\mathcal{F}(a)$ and elements of $\mathcal{F}(u)$ whose labels are related to a by the dependence relation. More precisely: $\leq_{(\mathcal{F}(u)\cdot\mathcal{F}(a))|_\Sigma} \;=\; (\vec{ua} \cap \Re_{\not<,\varphi_{ua}})^\star$.
- If $f = \mathcal{F}'$ then $\mathcal{F}'(ua)|_\Sigma \;=\; (\mathcal{F}'_\emptyset(u) \cdot \mathcal{F}'_p(a))\,|_\Sigma$ with $p = \mathcal{C}_u(\emptyset)$.
  Concatenation of $\mathcal{F}'_\emptyset(u)$ and $\mathcal{F}'_p(a)$ establishes comparabilities between the occurrence of letter a and elements of $\mathcal{F}'_\emptyset(u)$ that are dependent upon letter a: $\leq_{(\mathcal{F}'_\emptyset(u)\cdot\mathcal{F}'_p(a))|_\Sigma} = (\vec{ua} \cap \Re_{\not<,\varphi_{ua}})^\star$.

Which proves the lemma for words of size at most $k+1$. Finally the lemma is valid for the empty word.

Theorem 1 *Languages of P and $f(P)$ are linked by the following equality:*

$$[||P||]_\_ \;=\; \mathcal{L}\,(||f(P)||_\Sigma)$$

Proof With the help of lemma 1:

$$[||P||]_\_ \;=\; \bigcup_{u \in \|P\|} \mathcal{L}\,(\Gamma_{u,\not<})$$

Lemma 3 allows the compilation function f to be introduced:

$$[||P||]_\_ \;=\; \bigcup_{u \in \|P\|} \mathcal{L}\,(f(u)|_\Sigma) = \mathcal{L}\left(\left(\bigcup_{u \in \|P\|} f(u)\right)|_\Sigma\right)$$

We conclude with lemma 2.

$$[||P||]_\_ \;=\; \mathcal{L}\,(||f(P)||_\Sigma)$$

4 Synthesis of Distributed Programs

Automaton distribution is only partially achieved when the order-automaton has been produced. As a matter of fact, only dependence and synchronization problems are solved. Since the order-automaton codes for the behavior of an optimal distribution, it is now used to derive a distributed program (4.1). Its behavior should match the behavior of the order-automaton.

This distributed program is intended for a network of processors $I = 1 \ldots n$. Actually, since we are not concerned by the physical implementation, we will talk of processes rather than processors. The distribution is parametrized by the place where each action of alphabet Σ is assigned. This place is defined by the mapping $\pi : \Sigma \to I$.

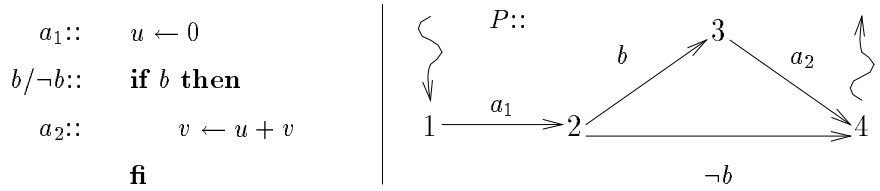
Synchronizations between actions placed on different processes are achieved by communications. These communications take place through a fully connected network of *FIFO* channels. This ultimate stage is explained throughout an example: the distribution of a sequential imperative program.

Finally the methodology of correctness proof of this stage is shortly described (4.2). It is based on the results of a previous work [3] which can be easily adapted to reactive systems.

4.1 Imperative Programs

4.1.1 Rationale

Let us now consider an imperative sequential program computing over a set of variables $\mathcal{V}$. The following example will be used throughout this section:



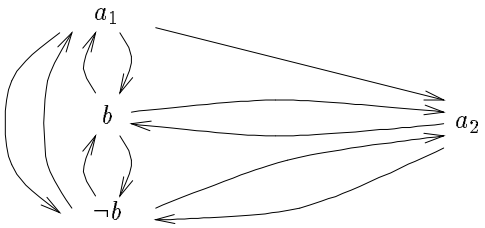
The control graph of this program can be represented by an automaton P. Its action alphabet Σ is the set of statements that compose the program. Each action is either a variable assignment or the predicate evaluation of a control statement (**if**, **while**, etc). For the example, $\Sigma = \{a_1, b, \neg b, a_2\}$.

The distribution is done on a data-driven basis: each variable is placed on a process. Accordingly, the set of variables $\mathcal{V}$ is partitioned over the network of processes I, i.e. into n subsets. For the example, $\mathcal{V} = \{u, b, v\}$, which is partitioned into $\mathcal{V}_1 = \{u\}$ and $\mathcal{V}_2 = \{b, v\}$.

All the generated processes execute identical code with respect to the control structure, but each assignment is only replicated on the process which owns the left-hand-side variable (*owner compute rule* [6]). In short, communication code is generated in order to implement access to remote variables: a copy of each referenced variable is to be sent by its owner towards the process that needs it [1, 3, 7]. However our only concern is distribution of control. Data exchange is not under consideration. Action $!S^a_{i\rightarrow j}$ (resp. $?S^a_{i\rightarrow j}$) is a message send (resp. receive) from process i to process j with value a. For all $i \in I$, $\Delta_i = \left\{?S^a_{j\rightarrow i}, !S^a_{i\rightarrow j}\right\}_{a\in\Sigma, j\in I}$ is the set of communication actions on process i. The set of all communication actions is denoted by $\Delta = \bigcup_{i\in I} \Delta_i$.

The partitioning of variables induces a partitioning of the action alphabet. The place where each predicate is evaluated can be chosen arbitrarily. The mapping $\pi : \Sigma \oplus \Delta \rightarrow I$ defines this partitioning. For the example, according to the data distribution, $\pi(a_1) = 1$, $\pi(a_2) = 2$, $\pi(b) = 2$, $\pi(\neg b) = 2$ and $\forall a \in \Sigma, \pi(!S^a_{1\rightarrow 2}) = 1$, etc.

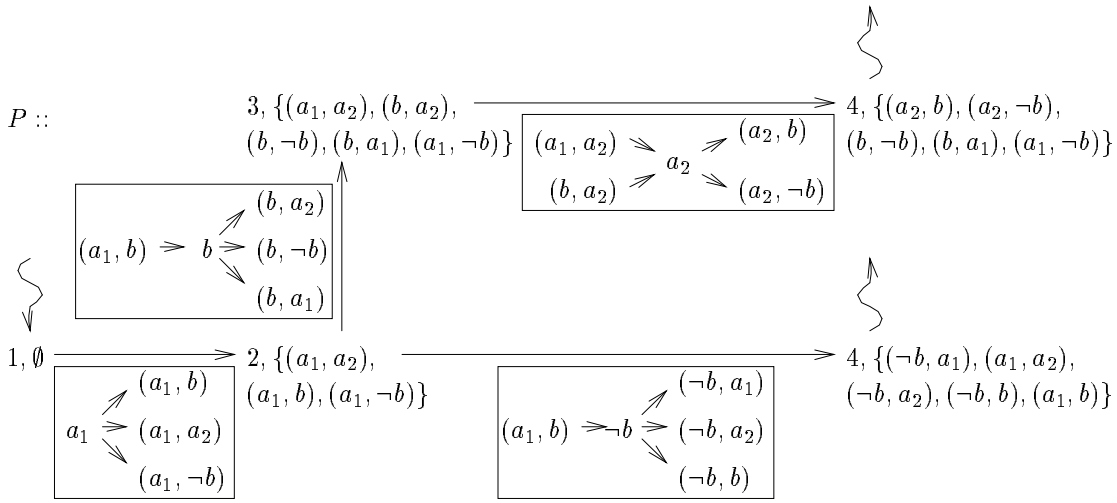
The independence relation $\propto$ follows the rules of data and control dependencies. Dependencies between assignments and communication actions are also considered. Actions placed on the same process are dependent. Predicate evaluations can not commute with any action: a total synchronization of the distributed system occurs each time a predicate is evaluated. For the example, the graph of the dependence relation is:



In order to keep the figure readable, dependencies with communication actions and self-dependencies are omitted.

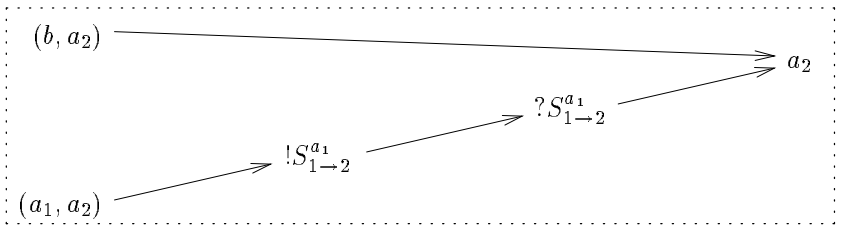
4.1.2 Distributed System Synthesis

Let us now consider the optimized order-automaton obtained from the automaton P and the independence relation $\propto$ (section 3.2). The task is now to synthesize a distributed program from this order-automaton:



The automaton P has some structural properties: it contains only binary choices and each choice is local to a process. The automaton has exactly one accepting state which is a maximal state (sink state). Most structural properties are preserved by order-automaton synthesis: the order-automaton contains only binary choices and sink accepting states.

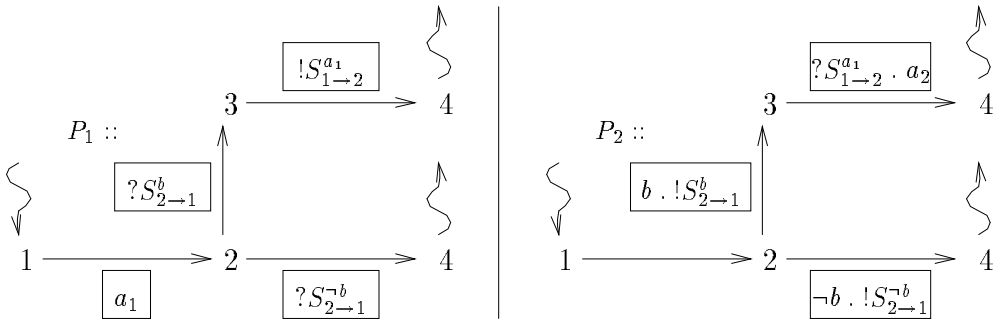
Communications are now inserted in the order-automaton. Basically, two matching dependencies linking actions belonging to two distinct processes are replaced by a message exchange. For instance, let us consider the transition from state 3 to state 4 of the example. There are two upstream dependencies, $(b, a_2) \rightarrow a_2$ and $(a_1, a_2) \rightarrow a_2$. They match respectively the two downstream dependencies $b \rightarrow (b, a_2)$ and $a_1 \rightarrow (a_1, a_2)$. As $\pi(b) = \pi(a_2) = 2$, the matching dependencies (b, a_2) lead to no communication at all. On the contrary, $\pi(a_1) = 1 \neq \pi(a_2)$, thus the matching dependencies (a_1, a_2) are turned into a communication from process $\pi(a_1)$, i.e. 1, to process $\pi(a_2)$, i.e. 2. Moreover, in order to ensure that each message sent is actually expected, we insert both send and receive actions where they are needed. Thus the upstream dependency $(a_1, a_2) \rightarrow a_2$ is turned into a send/receive:



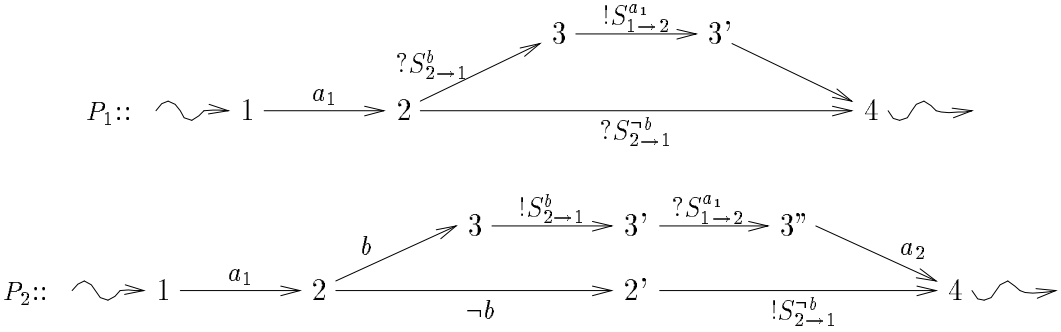
Beforehand, in order to ensure that messages are expected in the order of their sending, each transition label is replaced by one of its linear extensions. The particular choice of the linear extension is a matter of implementation. For the sake of clarity, linear extensions will be represented by words in $(\Sigma \oplus \Delta \oplus \not\propto)^\star$.

Then the order-automaton is projected on each process. For each $i \in I$, only actions that belong to process i are retained. Thus all transition labels are projected on the alphabet $\pi^{-1}(i)$. The projected order-automaton has transitions labeled by total orders. For the example,

$\pi^{-1}(1) = \{a_1, !S^{a_1}_{1\to 2}, ?S^{b}_{2\to 1}, ?S^{\neg b}_{2\to 1}\}$ and $\pi^{-1}(2) = \{a_2, b, \neg b, ?S^{a_1}_{1\to 2}, !S^{b}_{2\to 1}, !S^{\neg b}_{2\to 1}\}$. Thus the two projected order-automata are:

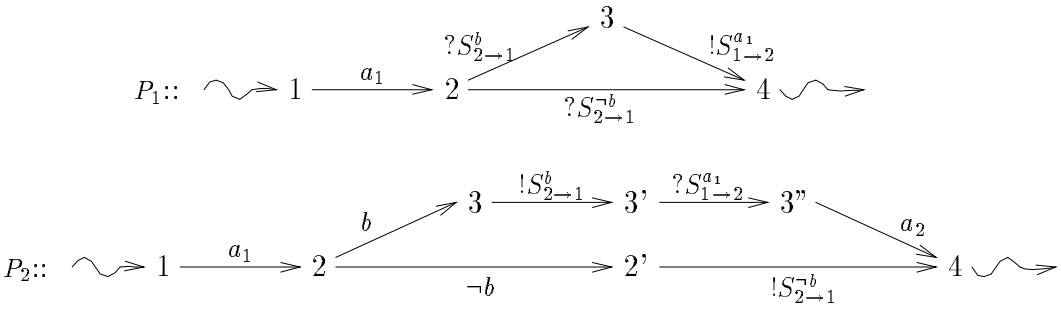


Such order-automata are easily transformed into non-deterministic automata over alphabets $\Sigma \oplus \Delta_1$ and $\Sigma \oplus \Delta_2$ respectively:



The ultimate stage of the distribution is the determinization of these automata. It should be noticed that (from a control point of view) only communications which correspond to a synchronization with a predicate evaluation should transmit a value.

With regard to the example, the final distributed program, where we have suppressed the transitions without any label, is:



4.2 Correctness Proof Issues

4.2.1 Imperative Programs

The distribution of a sequential imperative program produces a distributed system. It consists in a set of automata communicating through FIFO channels. The communication medium can be represented by a transition system, and the whole system is the product of these transition systems. The correctness proof of the transformation technique relies on the study of the properties of this product transition system. The proof is divided into two parts:

1. Distributed systems generated by this algorithm have the diamond property [3]. Hence the distributed system has exactly one maximal state unless there exists an infinite computation. In the latter case there is no finite maximal computation (Newman's theorem [2]).

2. Therefore it is sufficient to prove the correctness on a particular computation of the distributed system. This is done by comparing a behavior of the order-automaton with the causality order of this particular computation.

For further details, see [3] in which a detailed proof of a close result can be found.

4.2.2 Synchronous Reactive Programs

Synchronous reactive programs are not very different from sequential imperative programs. The only significant difference is the existence of communications with the environment. In other respects, there are internal variables and control statements, as in sequential imperative programs.

For all these reasons, the distribution of imperative and synchronous reactive programs is based on the same principles.

Communications with the environment are specific to reactive systems. They imply a major difference in the correctness proof of the distribution: infinite fair computation must be considered.

Consequently, concatenation of orders must be generalized to infinite orders. However difficult points are avoided because only fair computations are considered. Likewise, confluence must be generalized to infinite computations. This is achieved with the help of a measure on labeled partial orders. It gives a complete metric space structure to the set of labeled partial orders [13].

In spite of that, the principles of the correctness proofs are identical.

5 Conclusion

We have presented in this paper a formal approach to automata distribution. This approach is based on the synthesis of an order-automaton which codes the set of semantics preserving computations, modulo an independence relation. Then the order-automaton is mapped onto a distributed architecture and transformed into a set of communicating finite state machines.

This mapping is application dependent. An application is presented: the distribution of sequential imperative program.

The distributed system is considered as a product of labeled and partially deterministic transition systems, allowing the comparison of the behaviors accepted by the order-automaton and the possible behaviors of the distributed system. The combinatorics of the proof is mastered thanks to the diamond property of the generated distributed systems.

The outstanding unity of the fundamental problems encountered when considering two different distribution examples leads to a unified theory of distribution. The automaton distribution technique presented in the paper is a first attempt towards such a theory. In addition, our contribution may serve as a basis for designing and proving other (and new) parallelizing rules on several applications.

References

[1] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: a system to manage data distribution. In *ACM International Conference on Supercomputing*, June 11-15 1990.

[2] K.R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[3] C. Bareau, B. Caillaud, C. Jard, and R. Thoraval. Correctness of automated distribution of sequential programs. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, volume 694 of *LNCS*, pages 517–528. Springer Verlag, June 1993.

[4] G. Berry and A. Benveniste. The synchronous approach to reactive and real-time systems. *Another Look at Real Time Programming, Proceedings of the IEEE*, 79:1270–1282, 1991.

[5] G. Berry and G. Gonthier. Real time programming: special purpose or general purpose languages. In G. Ritter, editor, *Proc. IFIP congress*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989. Invited talk.

[6] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[7] P. Caspi and A. Girault. Distributing reactive systems. In *PDCS'94*. ISCA, October 1994.

[8] M. Clerbout and M. Latteux. Semi-commutations. *Information and computation*, 73:59–74, 1987.

[9] R. Cori and Y. Métivier. Recognizable subsets of partially abelian monoids. *Theoretical Computer Science*, 35:179–189, 1985.

[10] B.A. Davey and Priestley H.A. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[11] R.P. Hopkins. Distributable nets. In G. Rozenberg, editor, *Advances in Petri Nets 1991*, volume 524 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[12] C. Kant Antonescu. Synthèse de spécifications de protocoles à partir de spécifications de service. Thèse de doctorat, université de Montréal, mars 1993.

[13] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. An infinitary church-rosser property for non-collapsing orthogonal term rewriting systems. Technical Report CS-R9043, Centrum voor Wiskunde en Informatica, september 1990.

[14] A. Mazurkiewicz. Trace theory. In *Advanced Course on Petri Nets, LNCS # 255*, pages 279–324, 1986.

[15] A. Petit. Recognizable trace languages, distributed automata and the distribution problem. *Acta Informatica*, 30:89–101, 1993.

[16] K. Saleh and R. Probert. A service-based method for the synthesis of communications protocols. *International Journal of Mini and Microcomputers*, 12(3):97–103, 1990.

[17] W. Zielonka. Notes on finite asynchronous automata. *RAIRO Informatique Théorique et Applications*, 21(2):99–135, 1987.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399