Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors

Paul Caspi, Alain Girault, and Daniel Pilaud

Abstract—This paper addresses the problem of automatically distributing reactive systems. We first show that the use of synchronous languages allows a natural parallel description of such systems, regardless of any distribution problems. Then, a desired distribution can be easily specified, and achieved with the algorithm presented here. This distribution technique provides distributed programs with the same safety, test, and debug facilities as ordinary sequential programs. Finally, the implementation of such distributed programs only requires a very simple communication protocol ("first in first out" queues), thereby reducing the need for large distributed real-time executives.

Index Terms—Asynchronous communications, distributed processing, reactive systems, automatic distribution, synchronous languages.

1 Introduction

1.1 Reactive Systems

REACTIVE SYSTEMS are computer systems that react continuously to their environment, at a speed determined by the latter [18]. This class of systems contrasts, on one hand with *transformational systems* (classical programs whose inputs are available at the beginning of their execution and which deliver their outputs when terminating: for instance compilers), and on the other hand with *interactive systems* (which react continuously to their environment but at their own speed: for instance operating systems). Among reactive systems are most of the industrial real-time systems (control, supervision, and signal-processing systems), as well as man-machine interfaces. These systems have the main following features:

- **Parallelism**. At least, the design must take into account the parallelism between the system and its environment. Moreover, these systems are often implemented on parallel architectures, whether for reasons of performance increase, fault tolerance or functionality (geographical distribution). Finally, it is convenient and natural to design such systems as sets of parallel components that cooperate to achieve the intended behavior.
- **Determinism**. These systems always react in the same way to the same inputs. This property makes their design, analysis, and debugging easier. Thus, it should be preserved by the implementation.
- **Temporal requirements**. These requirements concern both the input rate and the input/output response time. They are induced by the environment and must imperatively be matched. Hence, they must be expressed in the specifications, they must be taken into account during the design, and their satisfaction must be checked on the implementation.
- **Reliability**. This is perhaps their most important feature as these systems are often critical ones. For instance, the consequences of a software error in an aircraft automatic pilot or in a nuclear plant controller are disastrous. Therefore, these systems require rigorous design methods as well as formal verification of their behavior.

A programming language well suited to the design of reactive systems should, therefore, be parallel and deterministic and allow formal behavioral and temporal verification.

1.2 The Synchronous Approach

Synchronous languages have been introduced in the '80s to make the programming of reactive systems easier [4]. The purpose of these languages is to give the designer ideal time primitives, thus reducing the chance of programming misconceptions. Instead of the interleaving paradigm, they are based on the simultaneity principle: All parallel activities share the same discrete time scale. Concretely, this means that $a \parallel b$ is viewed as the "package" ab where a and b are simultaneous. Each activity can then be dated on the discrete time scale; this has the following advantages:

- Time reasoning is made easier.
- Interleaving based nondeterminism disappears, which makes program debugging, testing, and validating easier.

Concerning the implementation, the idea is to project this discrete time scale onto the physical time. As the scale is discrete, *nothing* occurs between two consecutive instants: Everything must happen as if the processor running the program were infinitely fast. This is the *synchrony hypothesis*.

- *P. Caspi is with the Laboratoire VÉRIMAG, Centre Equation, 2 avenue de Vignate, 38610 Gières, France. E-mail: paul.caspi@imag.fr.*
- *A. Girault is with the Institut de Recherches en Informatique et Automatique (INRIA), Rhône-Alpes, Zirst, 655 avenue de l 'Europe, 38330 Montbonnot Saint-Martin, France. E-mail: alain.girault@inrialpes.fr.*
- *D. Pilaud is with POLYSPACE TECHNOLOGIES, co INRIA Rhône-Alpes, Zirst, 655 avenue de l'Europe, 38330 Montbonnot Saint-Martin, France. E-mail: daniel.pilaud@polyspace.com.*

Manuscript received 13 Sept. 1994; revised 11 Sept. 1997.
Recommended for acceptance by M. Young.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 105655.

Of course, such an infinitely fast processor does not exist, but it suffices that any input be treated before the next one. In order to verify this condition, one only needs to know the maximal input frequency, and an upper bound on the execution time of the object program. For this purpose, synchronous languages have deliberately restricted themselves to programs that can be compiled into a finite deterministic interpreted automaton, a control structure whose transitions are deterministic sequential programs operating on a finite memory. Each transition, whose execution time is statically computable, corresponds to the system reaction to an input.

There are numerous languages based upon the synchrony hypothesis: ESTEREL [5], LUSTRE [15], SIGNAL [20], STATECHARTS [17], SML [6], SYNCCHARTS [2], ARGOS [22], and SR [13]. Research on synchronous languages compilation has led to the Object Code (OC) encoding format for automata. It is the output format of the ESTEREL, LUSTRE, and ARGOS compilers [23].

For a better understanding of the synchrony hypothesis, let us study some examples in ESTEREL. ESTEREL is an imperative synchronous programming language. Besides variables, the language manipulates *signals*: A signal can be *valued* or *pure*, and can be an *input signal* (its presence can be tested), an *output signal* (it can be emitted), or a *local signal* (it can be emitted and its presence can be tested). The communication mechanism is the *synchronous broadcast*: any signal emitted by someone at a given instant is received by everybody at the same instant. Moreover, the temporal primitives of ESTEREL are intuitive, which will make the following examples easy to understand:

- Since control is passed instantly from a finishing statement to the next one, the statement `await 5 Second; await 5 Second` is equivalent to `await 10 Second`.[1]
- For the same reason, in the statement

```
every 60 MINUTE do
  emit HOUR;
end every;
```

  the signal `HOUR` is simultaneous with the 60th occurrence of the signal `MINUTE`.
- There is no notion of physical time inside a synchronous program, but rather an order relationship between events (simultaneity and precedence). The physical time is thus an external signal, like any other external signal. As a result, one can write either `abort TRAIN when 10 METER` or `abort TRAIN when 5 SECOND`.
- In the statement

```
  present A then
    % something
  end present;
||
  present B then
    % something else
  end present;
```

  each component of the parallel construct can react independently to its signal.[2] As a consequence, the program reacts either to `A` alone, `B` alone, or `A` and `B` at the same time.

1. The `await N S` statement waits for the `N`th occurrence of the signal `S`.
2. The statement `present S then P else Q` tests the presence of the signal `S` and has the same semantics as the statement if `E then P else Q`.

These small examples show that the synchrony hypothesis leads to very natural code. Providing the designer with ideal temporal primitives greatly reduces the number of programming errors. The drawback is that, once compiled, the execution time of the program must match the temporal specifications. But of course the same problem arises with an asynchronous programming language like ADA.

Finally, it is important to note that the synchronous approach has been validated through several real-life projects. Indeed, an industrial version of LUSTRE exists: It is the SCADE CASE tool developed by VÉRILOG. It is used by SCHNEIDER ELECTRIC for the control-command software of the nuclear plants, by AÉROSPATIALE for the flight control systems of the AIRBUS A340, as well as by 20 other companies in the transport and control-command industry. An industrial version of ESTEREL and SYNCCHARTS is also developed by SIMULOG: It is used by DASSAULT AVIATION for control systems of the RAFALE fighter.

1.3 Distribution Problems

Many reactive systems have to be distributed on several computing locations, for various reasons: performance increase, location of sensors and actuators, and fault tolerance. This is the case of the CO3N4 control system, developed at SCHNEIDER ELECTRIC for nuclear plants.

We consider here that distribution has to be specified by the system designer. There exist a priori three ways to achieve such a distribution:

1) **Compiling separately each piece of source program**, i.e., independently from its context, and making them communicate. This could be the ideal solution because it seems to be the easiest one. Unfortunately, Maffeïs [21] has shown that, in general, compiling separately pieces of programs into sequential deterministic programs is incorrect. However, Raymond [26] proposes some criteria for determining whether or not a piece of LUSTRE program is separately compilable. Also, ESTEREL gives criteria for compiling modules separately (cascade mode). On the other hand, separate compiling into nonsequential programs is always feasible: this is the SYNDEX solution presented in [19].
2) **Globally compiling a source program** into one sequential program for each location, so that each program may communicate with the others. This is the “abstract graph method” used for SIGNAL programs [21].
3) **Compiling the source program into a single object program**, and then distributing this centralized program towards as many programs as locations, so that each location only has to perform its own computations [8]. Based upon the common format OC, this method can be applied to any synchronous language.

The last two methods are complementary: The distribution of source programs avoids the problem of code size explosion, while the distribution of object programs offers the advantage of optimizing the centralized compiler and debugging the centralized object program before distributing it.

1.4 Distribution Method

The algorithm we present in this paper is based on the object code distribution method outlined in Fig. 1.

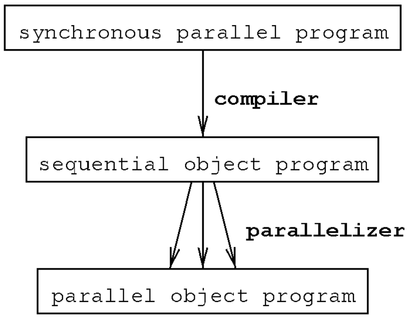


Fig. 1. Parallelization scheme.

Clearly, this approach raises the following question: Why not take advantage of the parallel aspects of the initial programs to directly synthesize communicating finite transition systems? We will not discuss this in full details and just list some reasons that justify the proposed approach:

- Parallelism in the synchronous languages aims at an easier and modular description of the system, and may not match the intended implementation parallelism.
- Compiling the program into a single transition system may be useful, in any case, for debugging and verification purposes [24], [11].
- Synchronous parallelism is not well-behaved for separate compiling matters. Thus, if communicating deterministic transition systems are desired, their direct synthesis may not be easier than the proposed method.

Our algorithm (described in Section 3) first duplicates the centralized OC program to make one copy for each location. It then removes in each copy the instructions not relevant to the current location, according to the distribution specifications provided by the user. At this point, the program of each location makes references to variables that are computed at a distant location. Our algorithm then adds communication instructions to each program to solve these data dependencies ([8], [3]). Finally, some problems like program resynchronization and redundant message elimination are addressed.

1.5 Paper Overview

Section 2 describes the OC format, the distribution specifications, and discusses the chosen communication primitives. Section 3 presents in full details the distribution algorithm. A small example is used to illustrate each steps. Also, for each step, the time and memory complexity are computed. Section 4 outlines the correctness proof of the distribution algorithm. Finally, Section 5 concludes and Section 6 shows some possible future research.

2 Preliminaries

Before describing the distribution algorithm, we present the OC format, the way for specifying a distribution, and the communication primitives we use.

2.1 The OC Format

A compiling method towards finite state automata has first been introduced for ESTEREL, and then adapted to LUSTRE and ARGOS.

Basically, the idea is to take advantage of the language determinism. It allows the building, at compile-time, of the tree of the program behaviors. This tree is generally infinite, but it can be folded into a finite automaton whose behavior is equivalent to the behavior of the program. This control automaton is associated with a finite memory for performing operations over infinite types.

The success of this method is guaranteed, first by the language determinism, and second by the static verifications performed beforehand. Finally, the language synchronism greatly reduces the explosion of the number of states. The benefits are:

- Usually, the automaton obtained is minimal.
- The equivalent program is purely sequential.
- The synchrony hypothesis can be easily checked.
- Several tools can be applied to the resulting automaton, for instance code generators, automaton minimizers, formal verification tools, visualizing tools, interface generators, and code distributors.

The automaton format used in compiling ESTEREL, LUSTRE, and ARGOS is the OC format. An OC program is a finite deterministic automaton with a finite memory for performing operations over infinite types. Basically, a program is a list of states, each containing some purely sequential code, represented by a Directed Acyclic Graph (DAG [1]). DAG actions are of two kinds:

- Control actions:
  - binary deterministic branchings: `if` (expression testing) and `present` (signal presence testing),
  - state change: `goto`.
- Sequential actions:
  - assignments to internal variables: `x := exp`,
  - signal emissions: `output(s)`,
  - external procedure calls: `foo(x, y)`.

Moreover, an OC program is a procedure which executes, each time it is called, *one* transition of the automaton. An interface is in charge of taking from the environment the inputs for the program, and calling the OC automaton procedure. In this execution scheme, inputs are updated by the program interface while outputs are emitted by the program itself.

2.2 Specifying a Distribution

The distribution specifications must result in the localization of each action on a location. Of course this localization must be unique and unambiguous. However, the problem of achieving the best localization will not be addressed in this paper.

At the source program level, we may assign for instance a location to each input and output signal of the program. By propagation, a location can then be assigned to each variable of the program.

At the OC level, we can directly assign a unique location to each variable of the program.

2.3 Choosing Communication Primitives

Finally, some form of communication and synchronization mechanism remains to be chosen. Shared variables do not allow synchronization between parallel processes, unless some complex mechanism is built on top of them. Moreover, they make formal verification harder. The other solution is message passing. Message passing in distributed systems can be synchronous or asynchronous [12]:

- Asynchronous message passing never blocks the sender. This requires an unbounded buffer; in practice, a bounded buffer is used and the sender will block when the buffer is full. Because the sender never has to wait, a higher degree of parallelism can be achieved. Moreover, sending statements can be moved backward while receiving statements can be moved forwarded, therefore minimizing the waiting time induced by the communication network.
- Synchronous message passing uses no buffer, so both senders and receivers can block. In this sense it leads to useless waiting times and reduces parallelism. The rendezvous used by classical real-time languages (ADA, OCCAM, and so on) are a form of synchronous message passing.

We choose asynchronous message passing in the form of two FIFO queues for each pair of locations, one in each direction. This is quite cheap in terms of execution environment. We define the two following communication actions:

- a sending action: `put(destination, value)` where `destination` is the location towards which the sending is done; `put`s are nonblocking;
- and a receiving action: `variable := get(source)` where `source` is the location from which the receiving action is done; `get`s are blocking when the queue is empty.

We, furthermore, require that the network preserve the ordering and the integrity of messages. This will ensure that values are not mixed up, provided that sending actions are inserted on one location in the same order as the corresponding receiving actions in the other location. For instance, assume that location 0 sends successively values 4 and 5 to location 1, and that location 1 must assign value 4 to variable `x` and value 5 to variable `y`. On location 0 we have the action `put(1, 4)` followed later by `put(1, 5)`. Inserting the actions `x := get(0)` and `y := get(0)` in that order on location 1 will ensure that the values are transmitted correctly.

3 THE DISTRIBUTION ALGORITHM

In each state of the automaton, the code is purely sequential. For simplicity, our distribution algorithm will operate at the state level. It consists of five steps, which we present successively:

1) replication and localization,
2) insertion of sending statements (`put`s),
3) insertion of receiving statements (`get`s),
4) synchronization of distributed programs,
5) elimination of redundant emissions.

3.1 Notations

In the sequel, we consider the following predicates:

- An action *belongs* to a location if this location must compute this action.
- A variable *belongs* to a location if this location locally computes this variable. Equivalently, we say that the location *owns* the variable.
- A location *needs* a variable if this location must compute an action that uses this variable (as a right-hand value of an assignment or in a branching).

3.2 Example

We illustrate the algorithm steps with the following OC program, for which we only give the code of state 0:

```
state 0
if (x) then
      y := x;
      output(y);
else
      x := true;
      y := x;
      output(y);
endif
goto state 1;
```

This program will be distributed onto two locations, according to the following specifications:

Location 0	Location 1
`y`	`x`

3.3 Replication and Localization

The problem is to assign a location to each action of the program. Based upon the distribution specifications, a unique location can be assigned to each variable of the program. The localization algorithm consists then, for each action, in building the list of locations that have to compute it:

- a control action (`if`, `present`, or `goto`): each location,
- a variable assignment: the location that owns the assigned variable,
- a signal emission: the location specified by the distribution.

As a consequence, the control is replicated on each location, i.e., each piece of program resulting from the distribution will have the same control structure. For our example, we have the following replication:

Location	state 0
(0, 1)	`if (x) then`
(0)	`      y := x;`
(0)	`      output(y);`
(0, 1)	`else`
(1)	`      x := true;`
(0)	`      y := x;`
(0)	`      output(y);`
(0, 1)	`endif`
(0, 1)	`goto state 1;`

For each action, we have indicated the list of locations that must perform it, i.e., the list of locations that own this action.

3.4 Insertion of Sending Statements (`puts`)

We perform the `put` insertion separately on each state of the automaton. In each state, we have a DAG whose nodes are the actions, and whose leaves are the `goto`s. The algorithm consists in associating with each location `s`, a set $\texttt{Need}_s$ of all the variables that location `s` will certainly need, provided that their value has not previously been sent by their owning location. The computation of the $\texttt{Need}_s$ sets allows the insertion of `put`s so that any location that needs a variable for a given action will receive it *before* the concerned action.

We propose the two following strategies:

- The *when needed* strategy where each variable is sent at the very moment when the destination location needs it. This will minimize the number of messages exchanged between two locations.
- The *as soon as possible* strategy where each variable is sent just after its computation on its owning location. This will increase the delay between the time when a value is sent and the time when it is needed by the destination location, and therefore shorten the waiting time on the destination location (remember that the `get` is blocking when the queue is empty).

A more precise comparison of the two strategies will be given in Section 3.6.

3.4.1 The When Needed Strategy

For each location `s`, the algorithm consists in placing an empty set $\texttt{Need}_s$ at each leaf of the DAG, and then propagating these sets backward to the root of the DAG in the following way:

- When reaching an action belonging to location `s`, if for this action, location `s` needs a variable `x` that belongs to another location, then add `x` to $\texttt{Need}_s$ (note that branchings also need variables).
- When reaching an assignment `x := exp`, for each location `s` such that $\texttt{x} \in \texttt{Need}_s$, insert the statement `put(s, x)` just after this assignment. Then remove `x` from each concerned set $\texttt{Need}_s$.
- When reaching a branching closure, duplicate the sets $\texttt{Need}_s$, and proceed in each branch `then` and `else`.
- When reaching a branching `if` or `present`, for each location `s`:
    - build the intersection of sets $\texttt{Need}_s^{\texttt{then}}$ and $\texttt{Need}_s^{\texttt{else}}$ from branches `then` and `else`;
    - in each branch, insert an action `put(s, x)` for each variable `x` of the set $\texttt{Need}_s - (\texttt{Need}_s^{\texttt{then}} \cap \texttt{Need}_s^{\texttt{else}})$; thus, a variable is sent when the target location needs it instead of when it is computed;
    - proceed with the intersection $\texttt{Need}_s^{\texttt{then}} \cap \texttt{Need}_s^{\texttt{else}}$.
- When reaching the root of the DAG, for each location `s`, insert at the beginning of the DAG a statement `put(s, x)` for each variable `x` of the set $\texttt{Need}_s$.

For our example, we obtain the following `put` placement:

location	state 0	$\texttt{Need}_0$	$\texttt{Need}_1$	
(1)	`put(0,x);`	∅	∅	③
(0,1)	`if (x) then`	{x}	∅	
(1)	`        put(0,x);`	∅	∅	②
(0)	`        y:=x;`	{x}	∅	
(0)	`        output(y);`	∅	∅	
(0,1)	`else`			
(1)	`        x:=true;`	∅	∅	
(1)	`        put(0,x);`	∅	∅	①
(0)	`        y:=x;`	{x}	∅	
(0)	`        output(y);`	∅	∅	
(0,1)	`endif`	∅	∅	
(0,1)	`goto state 1;`	∅	∅	

The algorithm has inserted three puts:

- the `put(0, x)` number ① because $\texttt{x} \in \texttt{Need}_0$ and `x` is modified by location 1;
- the `put(0, x)` number ② because $\texttt{x} \in \texttt{Need}_0^{\texttt{then}} - (\texttt{Need}_0^{\texttt{then}} \cap \texttt{Need}_0^{\texttt{else}})$;
- the `put(0, x)` number ③ because $\texttt{x} \in \texttt{Need}_0$ and the root of the DAG has been reached.

3.4.2 The as Soon as Possible Strategy

The goal here is to insert each `put` just after the last computation of the variable to be tansmitted. The algorithm is the same as before, except when reaching a branching `if` or `present`: We must then proceed with the union of sets $\texttt{Need}_s^{\texttt{then}}$ and $\texttt{Need}_s^{\texttt{else}}$ instead of the intersection. Besides, there is no `put` to be inserted after a branching action any more.

For our example, we obtain the following `put` placement:

location	state 0	$\texttt{Need}_0$	$\texttt{Need}_1$	
(1)	`put(0,x);`	∅	∅	②
(0,1)	`if (x) then`	{x}	∅	
(0)	`        y:=x;`	{x}	∅	
(0)	`        output(y);`	∅	∅	
(0,1)	`else`			
(1)	`        x:=true;`	∅	∅	
(1)	`        put(0,x);`	∅	∅	①
(0)	`        y:=x;`	{x}	∅	
(0)	`        output(y);`	∅	∅	
(0,1)	`endif`	∅	∅	
(0,1)	`goto state 1;`	∅	∅	

The algorithm has inserted two `put`s:

- the `put(0, x)` number ① because $\texttt{x} \in \texttt{Need}_0$ and `x` is modified by location 1;
- the `put(0, x)` number ② because $\texttt{x} \in \texttt{Need}_0$ and the root of the DAG has been reached.

3.4.3 Complexity

For the time and memory requirements, we adopt the following notations:

- For any procedure p, $\mathcal{T}(p)$, and $\mathcal{M}(p)$ denote, respectively, its time and memory requirements.

- nb_{loc}, nb_{var}, and nb_{act} are, respectively, the number of locations, of variables, and of actions of the distributed program.
- av_{var} is the average number of variables belonging to a given location.

We assume that the implementation allows any set operation to be performed in $O(av_{var})$, for instance with bitstreams. Hence, the time requirement for **put** insertion is $O(av_{var})$ times the cost of the action graph traversal. Thus:

$$\mathcal{T}(\texttt{put}\ \text{insertion}) = O(nb_{act} \times av_{var})$$

The memory requirement is the cost of the **Need** sets. Thus:

$$\mathcal{M}\ (\texttt{put}\ \text{insertion}) = O(nb_{var} \times nb_{loc})$$

3.5 Insertion of Receiving Statements (gets)

As for the sendings, we perform the **get** placement successively on each state. Receivings remain to be inserted, so that the actions **x := get(s)** appear in the program of location **t** in the same order as the actions **put(t, x)** in the program of location **s**. The hypothesis on the network (Section 2.3) ensures that the values exchanged between two locations by means of a **put**/**get** will always correspond to the same variable on each side.

The algorithm consists in simulating at any time the content of the waiting queues. We define for each pair of locations (**t**, **s**) a queue $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ containing the variables belonging to location **t** that have been sent to location **s** and not yet received by it. Those variables will be placed in the queue in their sending order.

For each pair of locations (**t**, **s**), the algorithm consists in placing an empty queue $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ at the root of the DAG, and then propagating those queues forward to the leaves of the DAG in the following way:

- When reaching an action **put(s, x)** on location **t**, add **x** at the tail of the queue $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$.
- When reaching an action that belongs to location **s**, if for this action location **s** needs a variable **x** that belongs to another location **t**, then necessarily $\texttt{x} \in \texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$. So, extract the head **h** of the queue $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ and insert the statement **h := get(t)** on location **s**. Repeat until **x** is extracted. This ensures that variables are extracted from the queue exactly in the same order they were inserted in.
- When reaching a branching **if** or **present**, duplicate queues $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$ and proceed in each branch **then** and **else**.
- When reaching a branching closure, for each pair of locations (**t**, **s**):
  - build the largest common suffix $\texttt{Suff}_{\texttt{t}\triangleright\texttt{s}}$ of queues $\texttt{Fifo}^{\texttt{then}}_{\texttt{t}\triangleright\texttt{s}}$ and $\texttt{Fifo}^{\texttt{else}}_{\texttt{t}\triangleright\texttt{s}}$ from branches **then** and **else**; this common suffix contains the variables, sent by location **t** to location **s**, that are located at the tail of both queues, and thus that have been sent the most recently: remember that the aim is to insert the **get** as late as possible, in order to minimize the waiting time induced by the network;
  - build the queue $\texttt{Rem}^{\texttt{then}}_{\texttt{t}\triangleright\texttt{s}} = \texttt{Fifo}^{\texttt{then}}_{\texttt{t}\triangleright\texttt{s}} - \texttt{Suff}_{\texttt{t}\triangleright\texttt{s}}$, respectively, **else**;
  - in the **then** branch, empty $\texttt{Rem}^{\texttt{then}}_{\texttt{t}\triangleright\texttt{s}}$, and for each variable **h** extracted at the head of the queue, insert the statement **h := get(t)** on location **s**; respectively, **else**;
  - proceed with $\texttt{Suff}_{\texttt{t}\triangleright\texttt{s}}$.
- When reaching a leaf, for each pair of locations (**t**, **s**), empty the queue $\texttt{Fifo}_{\texttt{t}\triangleright\texttt{s}}$, and for each variable **h** extracted at the head of the queue, insert the statement **h := get(t)** on location **s**.

3.5.1 The When Needed Strategy

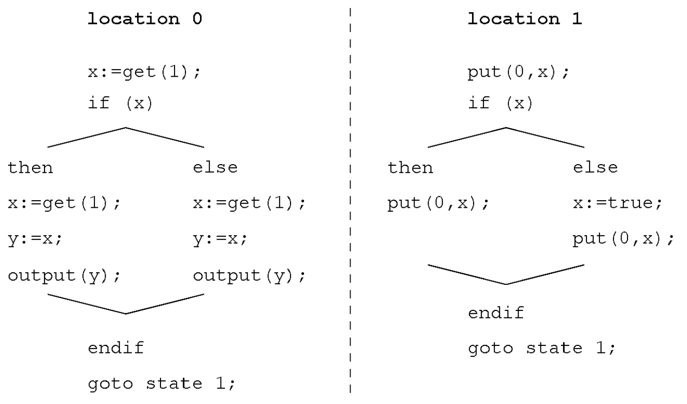
With our example where **put**s have been inserted with the *when needed* strategy, we obtain the following **get** placement:

location	state 0	$\texttt{Fifo}_{1\triangleright0}$	$\texttt{Fifo}_{0\triangleright1}$	
(1)	`put(0,x);`	$\cdots\texttt{x}]$	$\cdot\cdot\,]$	
(0)	`x:=get(1);`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	①
(0,1)	`if (x) then`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(1)	`    put(0,x);`	$\cdots\texttt{x}]$	$\cdot\cdot\,]$	
(0)	`    x:=get(1);`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	②
(0)	`    y:=x;`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(0)	`    output(y);`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(0,1)	`else`			
(1)	`    x:=true;`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(1)	`    put(0,x);`	$\cdots\texttt{x}]$	$\cdot\cdot\,]$	
(0)	`    x:=get(1);`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	③
(0)	`    y:=x;`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(0)	`    output(y);`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(0,1)	`endif`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	
(0,1)	`goto state 1;`	$\cdot\cdot\,]$	$\cdot\cdot\,]$	

The algorithm has inserted three **get**s:

- the **x := get(1)** number ① because $\texttt{x} \in \texttt{Fifo}_{1\triangleright0}$ and location 0 needs **x** to compute the branching **if (x)**;
- the **x := get(1)** number ② because $\texttt{x} \in \texttt{Fifo}_{1\triangleright0}$ and location 0 needs **x** to compute the assignment **y := x**;
- the **x := get(1)** number ③ because $\texttt{x} \in \texttt{Fifo}_{1\triangleright0}$ and location 0 needs **x** to compute the assignment **y := x**.

The final distributed program is shown in Fig. 2.



```
        location 0              |           location 1
                                |
        x:=get(1);              |           put(0,x);
        if (x)                  |           if (x)
                                |
then              else          |   then              else
x:=get(1);        x:=get(1);    |   put(0,x);         x:=true;
y:=x;             y:=x;         |                     put(0,x);
output(y);        output(y);    |
                                |           endif
        endif                   |           goto state 1;
        goto state 1;           |
```

Fig. 2. OC program distributed on two locations.

3.5.2 *The As Soon As Possible Strategy*

With our example where `put`s have been inserted with the *as soon as possible* strategy, we obtain the following `get` placement:

location	state 0	$\texttt{Fifo}_{1\triangleright 0}$	$\texttt{Fifo}_{0\triangleright 1}$	
(1)	`put(0,x);`	$\cdots\boxed{x}$	$\cdot\cdot\boxed{.}$	
(0)	`x:=get(1);`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	①
(0,1)	`if (x) then`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0)	`    y:=x;`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0)	`    output(y);`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0,1)	`else`			
(1)	`    x:=true;`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(1)	`    put(0,x);`	$\cdots\boxed{x}$	$\cdot\cdot\boxed{.}$	
(0)	`    x:=get(1);`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	②
(0)	`    y:=x;`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0)	`    output(y);`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0,1)	`endif`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	
(0,1)	`goto state 1;`	$\cdot\cdot\boxed{.}$	$\cdot\cdot\boxed{.}$	

The algorithm has inserted two `get`s:

- the `x := get(1)` number ① because $\texttt{x} \in \texttt{Fifo}_{1\triangleright 0}$ and location 0 needs `x` to compute the branching `if (x)`;
- the `x := get(1)` number ② because $\texttt{x} \in \texttt{Fifo}_{1\triangleright 0}$ and location 0 needs `x` to compute the assignment `y := x`.

The final distributed program is shown in Fig. 3

3.5.3 *Complexity*

We assume that the implementation allows the suffix computation to be performed in $O(av_{var})$, for instance with linked lists. Hence, the time requirement for `get` insertion is $O(av_{var})$ times the cost of the action graph traversal. Thus:

$$\mathcal{T}(\texttt{get}\text{ insertion}) = O(nb_{act} \times av_{var})$$

The memory requirement is the cost of the `Fifo` queues. Thus:

$$\mathcal{M}(\texttt{get}\text{ insertion}) = O(nb_{var} \times nb_{loc})$$

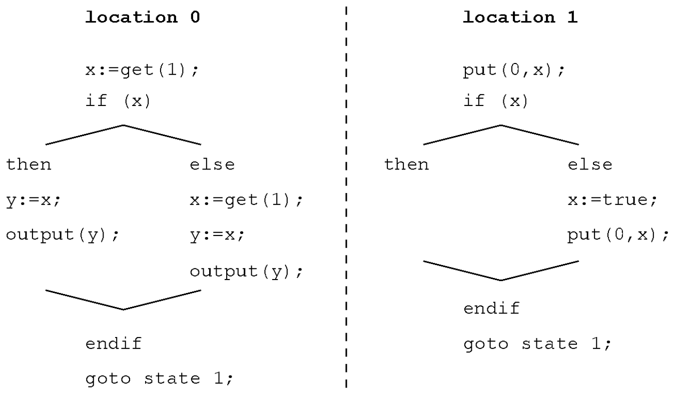
3.6 Comparison of the *When Needed and as Soon as Possible* Strategies

The *as soon as possible* strategy minimizes the waiting time on the receiving location. Indeed, for a given variable, the `put` is inserted just after the variable is computed, i.e., as soon as possible, while the `get` is inserted before the variable is used, i.e., as late as possible. However, when a variable is needed only in one branch of a test, then there will be a useless communication in the other branch. On the contrary, the *when needed* strategy minimizes the number of messages but leads to longer waiting times, since the `get` statement is blocking when the queue is empty.

Let us consider the following OC program:

state 0

`y := 10;`
`if (c) then`
`    x := y;`
`else`
`    y := y - 1;`
`endif`
`goto state 1;`



```
location 0                      location 1

x:=get(1);                      put(0,x);
if (x)                          if (x)

then          else              then          else
y:=x;         x:=get(1);                      x:=true;
output(y);    y:=x;                           put(0,x);
              output(y);
                                endif
endif                           goto state 1;
goto state 1;
```

Fig. 3. OC program distributed on two locations.

We decide to distribute it on two locations, with the following specifications: `y` belongs to location 0, while `x` and `c` belong to location 1. After the localization, replication, and insertion of `put` and `get`, we have:

when needed	
location	state 0
(1)	`put(0, c);`
(0)	`y := 10;`
(0)	`c := get(1)`
(0, 1)	`if (c) then`
(0)	`    put(1, y)`
(1)	`    y:= get(0)`
(1	`    x:= y`
(0, 1)	`else`
(0)	`    y := y - 1;`
(0, 1)	`endif`
(0, 1)	`goto state 1;`

as soon as possible	
location	state 0
(1)	`put(0, c);`
(0)	`y := 10;`
(0)	`put(1, y)`
(0)	`c: = get(1);`
(0, 1)	`if (c) then`
(1)	`    y := get(0);`
(1)	`    x := y:`
(0, 1)	`else`
(0)	`    y := y - 1`
(1)	`    y := get(0);`
(0, 1)	`endif`
(0, 1)	`goto state 1;`

The value of `c` is exchanged in the same way whatever be the chosen strategy. The `put` is made as soon as possible, i.e., just after the updating of `c`, which is performed implicitly at the beginning of the state because it is an input. The `get` is performed as late as possible, i.e., just before `c` is used, in the branching `if (c)`.

Concerning the value of `y`, it depends on the chosen strategy:

- The *when needed* strategy: The value of `y` is only exchanged in the `then` branch.

- The *as soon as possible* strategy: The value of `y` is exchanged in both branches, even though it is not needed in the `else` branch. The `put` statement is performed as soon as possible, i.e., just after the computation of `y`. The `get`s are performed as late as possible, i.e., just before `y` is used in the `then` branch, and just before the branching closure in the `else` branch. Moreover, this useless message cannot be suppressed because the `put` is performed before the branching, and, as a consequence, a `get` must be performed in each branch.

Finally, when a variable value is sent before a branching, and then modified by its owner in one of the branches while its value is needed in both branches, then the *when needed* strategy inserts a useless communication in the branch where the variable is not computed. So it seems that, with this strategy, the number of messages is not minimal either. Yet the difference with the *as soon as possible* is that the useless messages are redundant (i.e., the value exchanged is already known by the receiving location) and can be removed using classical static analysis techniques. This will be shown in Section 3.8.

3.7 Synchronization of Distributed Programs

Now it results from our distribution algorithm that some locations can behave as value producers while others behave as value consumers. In our program example (Fig. 2 and Fig. 3), location 0 is purely a consumer while location 1 is purely a producer. Thus, the program of location 1 may run faster than the program of location 0. This may lead to the loss of the centralized program temporal semantics and, since the `put` is never blocking, to unbounded queues at execution. We propose several solutions to achieve the resynchronization of distributed programs:

1) Use only bounded queues: a `put` will be blocking when the considered queue is full.
2) Add dummy communications so that there are no pure producers anymore.

The first method is easy, yet expensive, as it needs to check the status of the queue at each emission. Therefore, we choose the second method.

To achieve it, we introduce two dummy communication primitives, i.e., communications carrying no value: `put_void(destination)` and `get_void(source)`. This method must be applied on DAGs where only emissions have been inserted. It consists in adding dummy emissions. Then, `get`s and `get_void`s will be inserted at the same time directly on synchronized DAGs.

We have two options:

- **Strong synchronization:** Allow no cycle overlap between any two locations (Fig. 4a). Thus, no process may start its $n + 1$st reaction before all the others have terminated their nth reaction. To ensure this, we can add $(n - 1)$ synchronization messages (each message is one `put` and one `get`) at the end of each location program, for a total of $n \times (n - 1)$ synchronization messages. Another possibility consists in making a token circulate twice (the first time to make sure that everybody has completed its cycle, and the second time to permit each process to start its new cycle), which requires to add $2 \times n$ synchronization messages. The circulating token involves fewer messages but more execution overhead.
- **Weak synchronization:** Do not allow more than one cycle of overlap between any two locations (Fig. 4b). Thus, no process may start its $n + 2$nd reaction before all the others have terminated their nth reaction. This is much less expensive because the normal communications participate in the synchronization. To ensure weak synchronization, there must be, in each state, at least one communication in each direction between any pair of locations.

Adding messages for the strong synchronization is straightforward. On the other hand, the weak synchronization requires some flow analysis on the DAG of each state. To achieve that, we compute in each state and for each location `s`, the sets $\texttt{Lout}_s$ of locations towards which `s` has made no emission. For each location `s`, the algorithm consists in placing a full set $\texttt{Lout}_s$ at each leaf of the DAG, and then propagating these sets backward to the root of the DAG in the following way:

- When reaching the root of the DAG, insert a statement `put_void(t)` for each location `t` belonging to $\texttt{Lout}_s$.

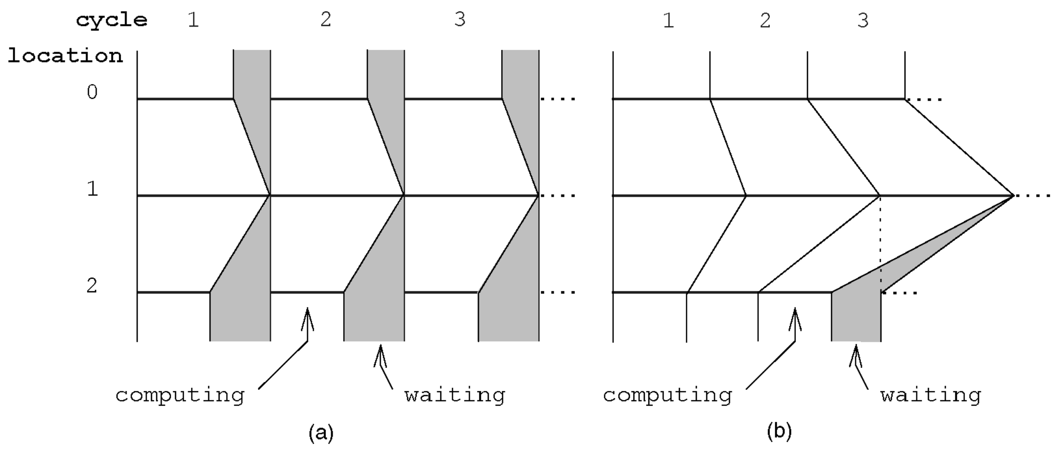


Fig. 4. Strong and weak synchronization.

- When reaching a branching closure, duplicate sets $\texttt{Lout}_s$, and proceed in each branch **then** and **else**.
- When reaching a branching **if** or **present**:
  - insert in branch **then** (respectively, **else**) a statement **put_void(t)** for each location **t** belonging to $\texttt{Lout}_s^{\texttt{then}} - \texttt{Lout}_s^{\texttt{else}}$ (respectively, $\texttt{Lout}_s^{\texttt{else}} - \texttt{Lout}_s^{\texttt{then}}$);
  - remove each location **t** for which we have inserted a **put_void(t)** statement in branch **then** (respectively, **else**) from set $\texttt{Lout}_s^{\texttt{then}}$ (respectively, $\texttt{Lout}_s^{\texttt{else}}$);
  - at this point, $\texttt{Lout}_s^{\texttt{then}}$ and $\texttt{Lout}_s^{\texttt{else}}$ are identical, so we proceed with $\texttt{Lout}_s^{\texttt{then}}$ or equivalently with $\texttt{Lout}_s^{\texttt{else}}$.
- When reaching the root of the DAG, insert a statement **put_void(t)** for each location **t** belonging to $\texttt{Lout}_s$.

The time and memory requirement are similar to those of the **put** insertion:

$$\mathcal{T}(\text{weak synchronization}) = O(nb_{act} \times nb_{loc})$$

$$\mathcal{M}(\text{weak synchronization}) = O(nb_{loc}^2)$$

For our program example, we obtain:

location	state 0	$\texttt{Lout}_0$	$\texttt{Lout}_1$	
(0)	`put_void(1);`	$\emptyset$	$\emptyset$	①
(1)	`put(0,x);`	{1}	$\emptyset$	
(0,1)	`if (x) then`	{1}	$\emptyset$	
(1)	`    put(0,x);`	{1}	$\emptyset$	
(0)	`    y:=x;`	{1}	{0}	
(0)	`    output(y);`	{1}	{0}	
(0,1)	`else`			
(1)	`    x:=true;`	{1}	$\emptyset$	
(1)	`    put(0,x);`	{1}	$\emptyset$	
(0)	`    y:=x;`	{1}	{0}	
(0)	`    output(y);`	{1}	{0}	
(0,1)	`endif`	{1}	{0}	
(0,1)	`goto state 1;`	{1}	{0}	

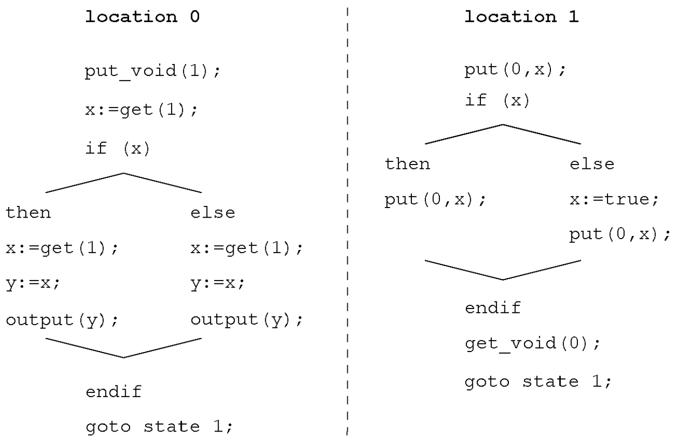
The algorithm has inserted one **put_void**:

- the **put_void(1)** number ① on location 0 because $1 \in \texttt{Lout}_0$.

After inserting the **get**s (Section 3.5) we obtain the final program of Fig. 5.

3.8 Elimination of Redundant Emissions

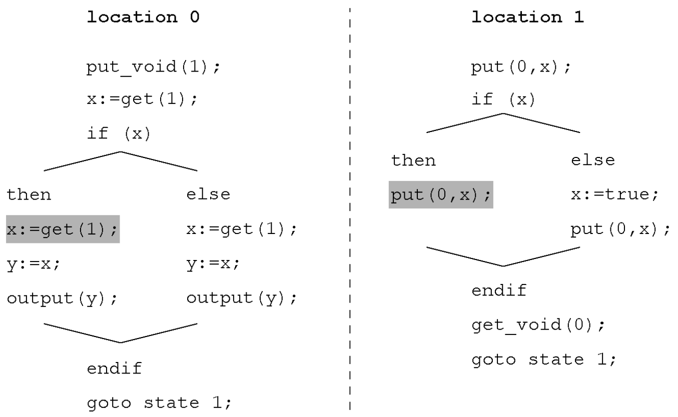
Now the **put** placement procedure sometimes causes redundant value emission (see Section 3.6). This occurs when a variable value is sent before a branching, and then is modified by its owner in one of the branches while its value is needed in both branches: then the *when needed* strategy inserts a redundant communication in the branch where the variable was not computed. In our example program, it is the case of the **put** / **get** in the **then** branch, as shown in Fig. 6.



```
location 0                              location 1

put_void(1);                            put(0,x);
x:=get(1);                              if (x)
if (x)                               then          else
then          else                   put(0,x);     x:=true;
x:=get(1);    x:=get(1);                           put(0,x);
y:=x;         y:=x;                         endif
output(y);    output(y);                    get_void(0);
      endif                                 goto state 1;
      goto state 1;
```

Fig. 5. OC program distributed on two locations and well synchronized.



```
location 0                              location 1

put_void(1);                            put(0,x);
x:=get(1);                              if (x)
if (x)                               then          else
then          else                   put(0,x);     x:=true;
x:=get(1);    x:=get(1);                           put(0,x);
y:=x;         y:=x;                         endif
output(y);    output(y);                    get_void(0);
      endif                                 goto state 1;
      goto state 1;
```

Fig. 6. OC program distributed on two locations with a redundant message.

However, since such communications are redundant, they can be eliminated using classical static analysis techniques. We show briefly how this can be achieved (the complete algorithm can be found in [14]).

We compute, for each location **s** and in each state of the automaton, the set $\texttt{Known}_s$ of variables known at the beginning of the state, i.e., whose values have been previously sent to **s** and which have not been modified by their owning location since then. Then, for each location and in each state, we propagate forward these sets:

- When reaching an emission **put(s, x)**, if $\texttt{x} \in \texttt{Known}_s$, then withdraw this **put(s, x)**, else add **x** to $\texttt{Known}_s$.
- When reaching an assignment **x := exp**, remove **x** from sets $\texttt{Known}_s$ for each location **s** that does not own **x**.
- When reaching a branching **if** or **present**, duplicate sets $\texttt{Known}_s$, and proceed in each branch **then** and **else**.
- When reaching a branching closure, for each location **s**, proceed with $\texttt{Known}_s^{\texttt{then}} \cap \texttt{Known}_s^{\texttt{else}}$.

The time and memory requirement are similar to those of the **put** insertion:

$$\mathcal{T}(\texttt{put}\ \text{elimination}) = O(nb_{act} \times av_{var})$$

$$\mathcal{M}(\texttt{put}\ \text{elimination}) = O(nb_{var} \times nb_{loc})$$

We apply this algorithm on DAGs where only emissions have been inserted. Thus, `get`s will be inserted directly on minimized DAGs. For our program example, we obtain:

location	state 0	$\texttt{Known}_0$	$\texttt{Known}_1$	
(0)	`put_void(1);`	$\emptyset$	{void}	
(1)	`put(0,x);`	{x}	{void}	
(0,1)	`if (x) then`	{x}	{void}	
(1)	`put(0,x);`	{x}	{void}	①
(0)	`y:=x;`	{x}	{void}	
(0)	`output(y);`	{x}	{void}	
(0,1)	`else`			
(1)	`x:=true;`	$\emptyset$	{void}	
(1)	`put(0,x);`	{x}	{void}	
(0)	`y:=x;`	{x}	{void}	
(0)	`output(y);`	{x}	{void}	
(0,1)	`endif`	{x}	{void}	
(0,1)	`goto state 1;`	{x}	{void}	

The algorithm has removed one `put`:

- the `put(0, x)` number ① because $\texttt{x} \in \texttt{Known}_0$.

After inserting the `get`s (Section 3.5) we obtain the final program of Fig. 7.

3.9 Algorithm Steps

Finally, the algorithm steps take place as follow:

1) replication and localization
2) insertion of sending statements (`put`s),
3) synchronization of distributed programs (`put_void`s),
4) elimination of redundant emissions,
5) insertion of receiving statements (`get`s and `get_void`s).

Therefore, receiving statements are inserted only once, on an already synchronized and optimized distributed program.

Now since these steps are sequential, the global time and memory requirements are:

$$\boxed{\mathcal{T}(\text{distribution algorithm}) = O(nb_{act} \times av_{var})}$$

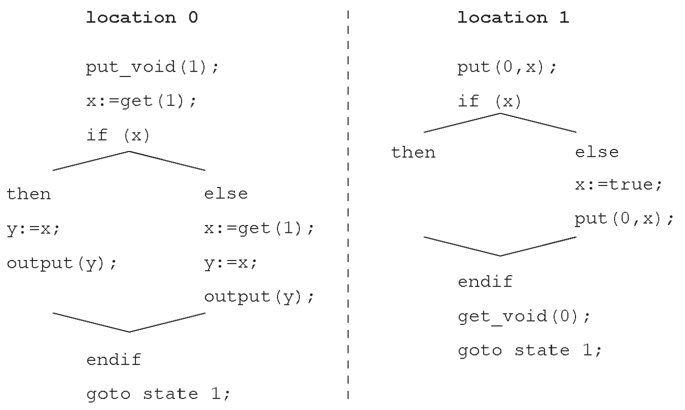
$$\boxed{\mathcal{M}(\text{distribution algorithm}) = O(nb_{var} \times nb_{loc})}$$

4 Correctness Proof

We have established in [7] the correctness proof of our distribution algorithm. We only outline the proof here. In order to prove that our distribution algorithm is sound, we have to prove that the behavior of the initial centralized program is equivalent to the behavior of the final parallel program.

We first model the initial centralized program by a finite deterministic automaton labeled with actions. Its behavior is the language of this automaton, i.e., the set of finite and infinite traces of actions it generates (trace semantics). The distribution specifications are given as a partition of the set of actions into *n* subsets, *n* being the number of intended computing locations.

We then define a *commutation relation* between actions according to the data dependencies. This commutation relation induces a *rewriting relation* over traces of actions. The set of all possible rewritings is the set of all admissible behaviors of the centralized program, with respect to the commutation relation. The problem is that this set cannot, in general, be recognized by a finite deterministic automaton. The intuition behind our proof is that this set is identical to the set of *linear extensions* of some partial order. For this reason we introduce a new model based on partial orders.



```
location 0                      location 1

put_void(1);                    put(0,x);
x:=get(1);                      if (x)
if (x)                          then        else
then        else                            x:=true;
y:=x;       x:=get(1);                      put(0,x);
output(y);  y:=x;               endif
            output(y);          get_void(0);
endif                           goto state 1;
goto state 1;
```

Fig. 7. OC program distributed on two locations with no redundant emissions.

- First, we build a centralized *order automaton* by turning each action labeling the initial automaton into a partial order capturing the data dependencies between this action and the remaining ones. The language of our order automaton is the set of finite and infinite traces of partial orders it generates (trace semantics). By defining a concatenation relation between partial orders, each trace is then itself a partial order. Thus, the language of our order automaton is a set of finite and infinite partial orders. Our key result is that the set of linear extensions of all these partial orders is identical to the set of all admissible behaviors of the centralized program, with respect to the commutation relation.
- Second, we show that our order automaton can be transformed into a set of parallel automata, by turning the data dependencies between actions belonging to distinct locations into communication actions, and by projecting the resulting automaton onto each computing location. We prove that these transformations preserve the behavior of our order automaton.

This formally establishes that the behavior of the initial centralized program is equivalent to the behavior of the final parallel program. There remains to prove that safety properties satisfied by the centralized program are also satisfied by the parallel program. Such properties express the fact that something will never happen, or that a given statement will always hold: They are expressed as temporal formulæ linking input and output signals of the program. In the case of a synchronous program, the temporal evolution of a signal is represented by its values at different cycles [25], [16]. Indeed, according to the synchrony hypothesis, any two signals that are emitted at the same cycle are simultaneous. Therefore, to ensure that safety properties are preserved, it is necessary to strongly synchronize the parallel program, as shown in Section 3.7. Indeed, strong synchronization will preserve the global cycle of the program: Output signals that are emitted at the same cycle by the centralized program will still be emitted at the same global cycle by the parallel program, even though they belong to distinct locations and are not linked by data dependencies. This key property cannot be achieved by weak synchronization.

5 Conclusion

Synchronous languages allow reactive systems to be programmed while preserving their natural parallelism. The algorithm we have presented automatically produces a distributed sequential code from a centralized synchronous program. As the program is first compiled, debugged and tested on a centralized processor, this method allows the production of a distributed code with the same safety as a centralized code. Finally, the distributed programs we obtain only need a very simple protocol in order to communicate (FIFO queues).

This algorithm has been implemented in the `Ocrep` tool. It provides the user with various options for the `put` insertion, the synchronization and the `put` elimination steps. The set manipulations are implemented by bit-set operations for efficiency purposes. The `Ocrep` tool is available online at http://www.inrialpes.fr/bip/people/girault/Ocrep. It has been tested on various synchronous programs obtained from reactive as well as robotic systems. The overhead due to synchronization and message passing between the different locations of the distributed program is low. For instance, a tennis game has been automatically distributed onto two locations: the average load goes from 80 percent on a single SPARC station for the centralized version to 50 percent on two SPARC stations for the distributed one.

We have stated that an OC program needs an interface to react to its environment. Distributing the program implies that its interface must also be distributed. An interface distribution method can be found in [10].

Finally, the formal proof of the distribution algorithm has been established. It rests on the modeling of the initial centralized program by a finite deterministic automaton labeled with actions, and on the abstraction of its admissible behaviors by a commutation relation (see Section 4 and [7]).

6 Future Research

Until now, all the processes obtained share the same control structure, which is the same as in the initial program. A more complex algorithm based on observational equivalence and "on-the-fly" bisimulation can be found in [9], which allows local minimization of each distributed process by suppressing branchings (`if` and `present`) whose branches have the same observable behavior. This technique, which remains to be studied and proven, allows a controlled form of desynchronization of synchronous programs:

- a long duration task scheduled on a slow clock can be inserted inside a synchronous program;
- to distribute this program, the distribution specifications have to partition the set of inputs/outputs in two subsets: one containing only the slow variables, and one containing all the remaining variables (hence, this is a clock driven distribution);
- then the minimization algorithm produces, for the slow location, a desynchronized program that actually runs at the slow clock speed; provided that the pace of the slow clock is compatible with the duration of the long duration task, this leaves it enough time to complete;
- at last, the synchronization algorithm described in Section 3.7 can be applied to ensure that the distributed program remains loosely synchronized.

Secondly, distributed real-time executives are expected to provide important fault-tolerance facilities, such as recovery data storage, error detection and masking, backward and forward recovery, and dynamic system reconfiguration. In most cases, these functions are carefully isolated from application programs, and a lot of research is still to be done in order to apply the techniques presented in this paper to this kind of problem.

Thirdly, when only physical data are involved (i.e., there are no discrete events), it is possible to conceive a parallel application by just programming separate tasks that run at their own speed and communicate through a dedicated network. When conceiving one task, the outputs of the other tasks are viewed as inputs to the current one. The network implements shared memories which are updated separately by each task. This form of communication does not allow synchronization between tasks because values can be lost without noticing. However, if only physical data are exchanged, such loss of data seems to be acceptable. Actually, a loss means that a fresher data has been updated by the emitting task and read by the receiving task. However, when discrete events are involved, this approach still needs to be formally studied. In particular, it is unclear at what speed the tasks and the network need to be run. Indeed, from these speeds depends the correct communication of data between the tasks.

Acknowledgments

Part of this work has been published in the *ISCA Seventh International Conference on Parallel and Distributed Computing Systems*. This work has been partially supported by GRECO Automatique action C2A, Ministère de l'Enseignement Supérieur et de la Recherche, and SCHNEIDER ELECTRIC.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, June 1987.

[2] C. André, "Representation and Analysis of Reactive Behaviors: A Synchronous Approach," *CESA*'96, Lille, France, IEEE-SMC, July 1996.

[3] F. André, J.-L. Pazat, and H. Thomas, "PANDORE: A System to Manage Data Distribution," *Int'l Conf. Supercomputing*. ACM, June 1990.

[4] G. Berry and A. Benveniste, "The Synchronous Approach to Reactive and Real-Tme Systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1,270–1,282, Sept. 1991.

[5] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[6] M.C. Browne and E.M. Clarke, "SML: A High-Level Language for the Design and Verification of Finite State Machines," *Int'l*

Working Conf. from HDL Descriptions to Guaranteed Correct Circuit Designs, Grenoble, France, IFIP, Sept. 1986.

[7] B. Caillaud, P. Caspi, A. Girault, and C. Jard, “Distributing Automata for Asynchronous Networks of Processors,” *European J. of Automation (RAIRO-APII-JESA)*, vol. 31, no. 3, pp. 503–524, 1997.

[8] D. Callahan and K. Kennedy, “Compiling Programs for Distributed Memory Multiprocessors,” *J. Supercomputing*, vol. 2, pp. 151–169, 1988.

[9] P. Caspi, J.-C. Fernandez, and A. Girault, “An Algorithm for Reducing Binary Branchings,” P.S. Thiagarajan, ed., *Proc. 15th Conf. Foundations of Software Technology and Theoretical Computer Science, FST & TCS’95*, Lecture Notes in Computer Science 1,026, Bangalore, India, Springer-Verlag, Dec. 1995.

[10] P. Caspi and A. Girault, “Execution of Distributed Reactive Systems,” S. Haridi, K. Ali, and P. Magnusson, eds., *First Int’l Conf. Parallel Processing, EURO-PAR’95*, pp. 15–26, Stockholm, Sweden, Lecture Notes in Computer Science 966, Springer-Verlag, Aug. 1995.

[11] E.M. Clarke, E.A. Emerson, and A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications,” *ACM TOPLAS*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[12] A. Dinning, “A Survey of Synchronization Methods for Parallel Computers,” *Computer*, pp. 66–76, July 1989.

[13] S. Edwards, “The Specification and Execution of Heterogeneous Synchronous Reactive System,” thesis, Univ. of Calif., Berkeley, 1997.

[14] A. Girault, “Sur la Répartition de Programmes Synchrones,” thesis, INPG, Grenoble, France, Jan. 1994.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, “The Synchronous Data-Flow Programming Language LUSTRE,” *Proc. IEEE*, vol. 79, no. 9, pp. 1,305–1,320, Sept. 1991.

[16] N. Halbwachs, F. Lagnier, and C. Ratel, “An Experience in Proving Regular Networks of Processes by Modular Model Checking,” *Acta Informatica*, vol. 29, nos. 6/7, pp. 523–543, 1992.

[17] D. Harel, “STATECHARTS: A Visual Approach to Complex Systems,” *Science of Computer Programming*, vol. 8, no. 3, 1987.

[18] D. Harel and A. Pnueli, “On the Development of Reactive Systems,” *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag, 1985.

[19] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine, “The SYNDEX Software Environment for Real-Time Dstributed Systems Design and Implementation,” *Proc. European Control Conf.*, vol. 2, pp. 1,684–1,689. Hermes, July 1991.

[20] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire, “Programming Real-Time Applications with SIGNAL,” *Proc. IEEE*, vol. 79, no. 9, pp. 1,321–1,336, Sept. 1991.

[21] O. Maffeïs, “Ordonnancements de Graphes de Flots Synchrones; Application à la Mise en Oeuvre de SIGNAL,” thesis, Univ. of Rennes I, Rennes, France, Jan. 1993.

[22] F. Maraninchi, “Operational and Compositional Semantics of Synchronous Automaton Compositions,” W.R. Cleaveland, ed., *Proc. Third Int’l Conf. Concurrency Theory, CONCUR’92*, Lecture Notes in Computer Science 630, pp. 550–564, Stony Brook, Springer-Verlag, Aug. 1992.

[23] J.P. Paris et al., “Les formats communs des langages synchrones,” Technical Report 157, INRIA, June 1993.

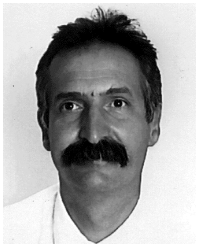
[24] J.-P. Queille and J. Sifakis, “Specification and Verification of Concurrent Systems in CESAR,” *Int’l Symp. Programming*, pp. 337–351, Lecture Notes in Computer Science 137, Springer-Verlag, Apr. 1982.

[25] C. Ratel, N. Halbwachs, and P. Raymond, “Programming and Verifying Real-Time Systems by Means of the Synchronous Data-Flow Language LUSTRE,” *IEEE Trans. Software Eng.*, vol. 18, no. 9, pp. 785–793, Sept. 1992.

[26] P. Raymond, “Compilation séparée de programmes LUSTRE,” DEA report, Joseph Fourier Univ., Research Report SPECTRE L5, Grenoble, France, June 1988.

Paul Caspi is docteur ès sciences and senior researcher at CNRS, the French National Research Center in Grenoble, France. His domain of interest is computer science applied to automatic control. He is mainly concerned with safety problems in critical applications from both hardware and software points of view. This has led him to be involved in the design of LUSTRE, a data-flow programming language for safety-critical automatic control applications. He also served as a consultant for several French companies and administrations on problems related to safety-critical computing systems. His present research domain deals with synchrony and the relationships between synchrony and asynchrony. This has led him to study distributed implementations of synchronous programs from both theoretical and practical points of view, as well as extensions of the synchronous framework toward dynamical (recursively defined) and higher order data-flow networks.

Alain Girault holds a research fellow position at INRIA, the Institut de Recherches en Informatique et Automatique in Grenoble, France. He received his PhD from the National Polytechnical Institute of Grenoble in 1994 and spent one year as a postdoctoral researcher in the ESTEREL team at INRIA Sophia-Antipolis. Dr. Girault also worked in the PTOLEMY and PATH groups at the University of California at Berkeley. His research interests include the design of reactive systems, with a special concern for distributed implementation and formal verification. He designed the `Ocrep` tool that parallelizes synchronous programs according to distribution specifications given by the user.



Daniel Pilaud received his PhD from the National Polytechnical Institute of Grenoble in 1982 on the design of real-time systems. From 1982–1989, he has been teacher-cum-researcher at ENSIMAG and participated in research work on the synchronous language LUSTRE. In 1989, he joined VÉRILOG and ran the technical center of Grenoble. Particularly, he was in charge of the development of SCADE environment, a programming environment for critical real-time software based on the LUSTRE language. The SCADE environment is used by leading companies in the areas of avionics, energy, and railway transport. In 1998, he left VÉRILOG to join INRIA and supervised the start up of a company that is in the field of control and validation of critical systems, POLYSPACE TECHNOLOGIES.