Automatic Rate Desynchronization of Embedded Reactive Programs

ALAIN GIRAULT
INRIA Rhône-Alpes, POP ART project, Montbonnot, FRANCE
and
XAVIER NICOLLIN
INPGrenoble, VERIMAG, Gières, FRANCE
and
MARC POUZET
Université Paris-Sud, LRI, Orsay, FRANCE

Many embedded reactive programs perform computations at different rates, while still requiring the overall application to satisfy very tight temporal constraints. We propose a method to automatically distribute programs such that the obtained parts can be run at different rates, which we call *rate desynchronization*. We consider general programs whose control structure is a finite state automaton and with a DAG of actions in each state.

The motivation is to take into account long duration tasks inside the programs: these are tasks whose execution time is long compared to the other computations in the application, and whose maximal execution rate is known and bounded. Merely scheduling such a long duration task at a slow rate would not work since the whole program would be slowed down if compiled into sequential code. It would thus be impossible to meet the temporal constraints, unless such long duration tasks could be *desynchronized* from the remaining computations. This is precisely what our method achieves: it distributes the initial program into several parts, so that the parts performing the slow computations can be run at an appropriate rate, therefore not impairing the global reaction time of the program.

We present in detail our method, all the involved algorithms, and a small running example. We also compare our method with the related work.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.2 [**Software**]: Programming languages—*Concurrent, distributed, and parallel languages*; D.3.2 [**Software**]: Programming languages—*Data-flow languages*

Author's address: A. Girault, INRIA Rhône-Alpes, POP ART project, 655 avenue de l'Europe, 38334 Saint-Ismier cedex, France, Email: `Alain.Girault@inrialpes.fr`
X. Nicollin, INPGrenoble, VERIMAG, Centre Équation, 2 avenue de Vignate, 38610 Gières, France, Email: `Xavier.Nicollin@imag.fr`
M. Pouzet, Université Paris-Sud, LRI, 91405 Orsay cedex, France, Email: `Marc.Pouzet@lri.fr`
A shorter version of this article has been published in the conference EMSOFT 2003 under the title "Clock-Driven Automatic Distribution of Lustre Programs".

General Terms: Algorithms, Design
Additional Key Words and Phrases: Embedded programs, reactive systems, desynchronization, automatic distribution, long duration tasks, parallelization algorithm

1. INTRODUCTION

1.1 Embedded reactive programs

Embedded programs are programs running on systems such as autonomous vehicles, mobile systems, satellites, and so on. Their main requirement is to comply with *limited resources*: limited memory, limited computing power, limited power supply... Often, they must also react continuously to their environment, at a speed determined by the latter, and are thus termed as *reactive programs* [Harel and Pnueli 1985]. As such, they must meet the following additional requirements:

(1) **Temporal requirements.** This concerns both the input rate and the input/output response time. To check their satisfaction on the implementation, it is necessary to know bounds on the execution time of each computation as well as on the maximal input rate.

(2) **Safety requirements.** Being intrinsically critical, they require rigorous design methods and languages as well as formal verification and validation of their behavior.

(3) **Parallelism requirements.** At least, the design must take into account the parallelism between the system and its environment. Moreover, these systems are sometimes implemented on distributed architectures, whether for reasons of processor load, performance increase, fault tolerance or functionality (geographical distribution).

1.2 Automatic rate desynchronization

In this paper, we present a method to achieve the *automatic rate desynchronization* of embedded programs. The programs we consider have a control structure expressed as a *finite state automaton*, and are meant to be run in an *execution loop*, where one iteration of this loop corresponds to one reaction of the program, and where inputs are obtained from the environment at the beginning of each state of the automaton. Therefore, a reaction of the program involves obtaining the inputs from the environment and then executing one transition of its automaton.

Given a centralized source program and some distribution specifications provided by the user, we will produce *automatically* a desynchronized program having the same observable behavior as the one of the centralized source program. This will consist of a distributed program, with as many parts as required by the distribution specifications. These parts will communicate harmoniously through asynchronous buffers, and each will have its own execution loop. By *rate desynchronization* we mean that the rates of the obtained part will not necessarily be all identical, and thus not identical to the rate of the centralized source program.

1.3 Motivation: long duration tasks

One of the main motivations for rate desynchronization is the handling of *long duration tasks*. These computations have the following three characteristics:

(1) their execution time is *long* compared with the other computations in the application,

(2) their execution time is *known* and *bounded*,

(3) their maximal execution rate is *known* and *bounded*.

Of course, the execution time and execution rate of a long duration task must be consistent with the performance of the hardware running the system. Otherwise the temporal requirements of the system will never be met. We will address this issue in § 5.1.

Many embedded reactive programs do involve long duration tasks. This is the case of the CO3N4 software control system, developed at Schneider Electric for nuclear plants [Bergerand and Pilaud 1988]. One of CO3N4's subsystems had a very tight timing constraint, and its first implementation could not meet it. This subsystem consisted of two parts, a slow one performing a long duration task, and a fast one performing everything else. The engineers came up with a solution where the slow part was cut into two subparts, with the system performing at each of its cycles the fast part, and alternately the first slow subpart and the second slow subpart.

As an example, consider a system with three tasks: task A performs slow computations (duration=8, period=deadline=32); task B performs medium and more urgent computations (duration=6, period=deadline=24); and finally task C performs the fastest and most urgent computations (duration=4, period=deadline=8). There are two ways to implement such a system:

(1) **Manual task slicing.** Tasks A and B are sliced into small chunks which are interleaved with task C, as shown in Figure 1 (the chunks are denoted by A1, A2, A3, and A4 for tasks A, and B1, B2, and B3 for task B). Such a manual task slicing is very hard to achieve, error prone, and difficult to debug. The main reasons are that the slicing itself is complex, in the general case various subparts may communicate with each other, and finally, obtaining a correct and deadlock-free interleaving is difficult.

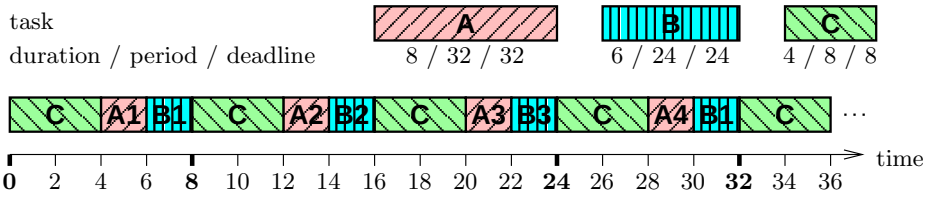

Fig. 1. Manual task slicing.

(2) **Distribution into three processes.** Here, tasks A, B, and C are performed by one process each, and the task slicing is done by the scheduler of the underlying real-time operating system, with some priority policy.[1] There are two ways to achieve such a distribution into three processes:
   (a) Manually programming three asynchronous tasks. The example of the Mars Rover Pathfinder[2] shows the limitation of this approach for critical systems. In the Mars Rover case, several parallel tasks had to share common resources. Since some tasks were more urgent than others, the designers used priorities. However, at some point during execution, the phenomenon known as priority inversion occurred [Sha et al. 1990], causing a total system reset! The intrinsic non determinism of the system, combined with the fact that each parallel subsystem was designed separately, made the problem all the more difficult to debug. We claim that this approach — manual programming of a real-time parallel program — makes the program harder to debug, test, and verify.
   (b) Distributing a centralized program into three processes. To achieve automatically a distributed implementation, we choose the so-called *object code distribution method*. It involves first building the complete object code of the centralized program (that is, a single task program), and then distributing it according to the system designer's specifications. The main advantage of this approach is that it is easier and less error-prone than to design directly a distributed system. This explains the recent success of automatic distribution methods (see [Gupta et al. 1999] for a survey of existing methods, or [Caspi et al. 1999] for a more specific algorithm working on programs having a finite state automaton control structure). The other advantage of this approach is the possibility to debug and formally verify the centralized program *before* its distribution, which is always easier and faster than debugging a distributed program. Finally, there remains the issue of the correctness of this approach: it will be addressed in § 3.1.

In other words, the solution we propose avoids, not only the manual partition of A and B into small chunks A_i and B_i (as in solution 1), but also the manual partition of the whole program into A, B, and C (as in solution 2a). Moreover, since the distribution will be automated, it will allow the user to test several partitions along with several priority policies.

1.4 Outline of the paper

The paper is organized as follows. In Section 2, we present the format of our programs. In Section 3, we present a first attempt of automatic distribution and we discuss why this does not work, i.e., why the rates of all the obtained parts are identical. In Section 4, we explain in detail our method for achieving the rate desynchronization. Finally, in Section 5, we discuss issues like worst case execution time, implementation, and related work, and we give some concluding remarks.

[1] This priority problem is *orthogonal* to the scope of this paper. For instance, one can choose the Rate Monotonic scheduling policy (RM [Liu and Layland 1973]), where the order relation between the task priorities is the reverse of the order relation between the task periods.
[2] This story report is available at `http://www.cs.cmu.edu/afs/cs/user/raj/www/mars.html`.

2. PRELIMINARIES

Our method for achieving the rate desynchronization of programs heavily uses past work on automatic distribution, based on a finite state automaton format called OC. In this section, we present first this format (basically a finite state automaton with a DAG of sequential actions in each state), and then an OC program that will serve as a running example in subsequent sections.

2.1 Automaton format

Our distribution algorithm [Caspi et al. 1999] uses an automaton format named OC (for Object Code [Plaice and Saint 1987]). An OC program is a finite deterministic automaton. This state graph can be cyclic, but in each state, there is sequential acyclic code, represented by a rooted binary directed acyclic graph (DAG) of actions. A program manipulates three kinds of variables: *input* variables can be used only as r-values; *local* and *output* variables can be used as r-values and l-values; *output* variables are also written to the environment.

Each DAG has one root (graphically represented by a circled dot), several unary and binary nodes, and one or more leaves:

—Unary nodes are sequential actions, which can be either:
  —an indication that the input variables of the program have been read and that their values are available in the local memory of the program: `go(...,in`$_i$`,...)`, where `in`$_i$ are the inputs; this `go` action must be the first node of each DAG, it makes explicit the interaction of the program with its environment;
  —an assignment to a local or output variable: `var:=exp`, where `exp` can contain external function calls;
  —an output writing: `write(var)`;
  —an external procedure call: `proc(...,var`$_i$`,...)(...,val`$_j$`,...)`, where `var`$_i$ and `val`$_j$ are respectively the variable and value parameters.
—Binary nodes are deterministic branchings: `if (var) then p else q endif`, where `p` and `q` are (possibly empty) subdags.
—Leaves, and only leaves, denote the next state number: `goto n`.

This automaton format is quite general since programs written in a classical imperative programming language can be compiled into this format. In fact, any OC program can be translated into a flow graph of basic blocks, and vice-versa.

Finally, concerning the execution, an OC program is embedded in an *execution loop*: at each cycle, the inputs are read from the environment, and then one transition of the automaton is executed (i.e., the code of the current state's DAG is executed). The outputs are written to the environment when executing the automaton's transition, through the `write` actions. For a reactive system, checking the temporal constraints amounts to verifying that the automaton can be run in an execution loop whose period is less than the maximal time allowed by the temporal constraint.

2.2 The `FILTER` OC program

We consider the following OC program `FILTER`. It has two states, numbered 0 and 1, 0 being the initial state. In each state, the DAG can be depicted as a graph

or equivalently as a textual list of instructions. Figure 2 below shows the textual representation of the FILTER program.

```
state 0

go(CK,IN);
if (CK) then {
  RES:=0;
  write(RES);
  OUT:=SLOW(IN);
  write(OUT);
  goto 1;
} else {
  write(RES);
  goto 0;
} endif;
```

```
state 1

go(CK,IN);
if (CK) then {
  RES:=OUT;
  OUT:=SLOW(IN);
  write(OUT);
} else {
} endif;
write(RES);
goto 1;
```

Fig. 2. Textual representation of the centralized FILTER program.

Figure 3 below shows the equivalent graphical representation of the FILTER program.

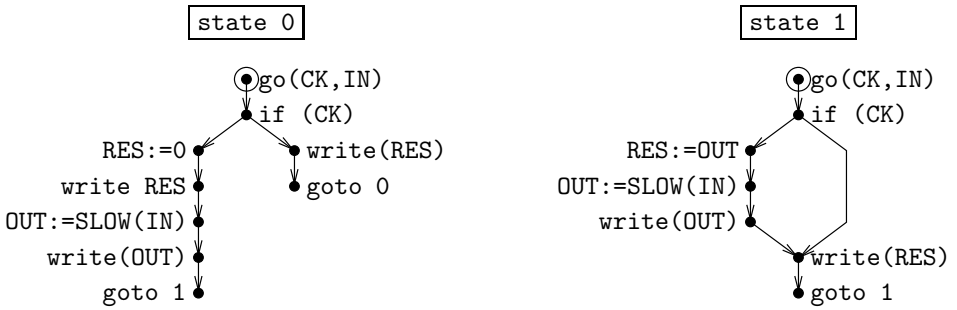

Fig. 3. Graphical representation of the centralized FILTER program.

The FILTER program has:

—two inputs: the Boolean CK and the integer IN;

—two outputs: the integers RES and OUT; OUT is computed by calling the external function SLOW; this function is supposed to perform a long and complicated computation; here, for sake of simplicity, we assume that the result of SLOW(n) is 3*n.

Additionally, each input and output has a *rate*, which is the sequence of cycles where it "exists", that is, where it bears a value. *Nothing* is known a priori about the rates; in particular, they do not need to be periodic. Here, IN is used only when CK is true, so we say that its rate is CK.[3] In contrast, since CK is used at

[3]Even though IN is *read* at each cycle, by the go action, it is *used* only when CK is true.

each cycle, we say that its rate is the *base rate* of the program. OUT is written each time CK is true, hence its rate is CK. Finally, RES is written at each cycle so its rate is the base rate.

Now, since the branchings can be nested, so can be the rates. Therefore, we can build a *tree of rates*, whose root is the base rate of the program and whose nodes are the different rates. A rate C1 is then said to be *faster* than another rate C2 if C2 is in the subtree whose root is C1. Each node of this tree can then be decorated with the set of inputs and outputs whose rate is precisely the node.

The FILTER program remains in state 0 until the first instant when CK bears the value true; then it moves to state 1. Figure 4 below is an example of a run. In this run, the rate CK is periodic (true at instants 1, 4, 7, and so on, as shown by the thick bars above), but this is not necessary the case, and it is not required by our method.

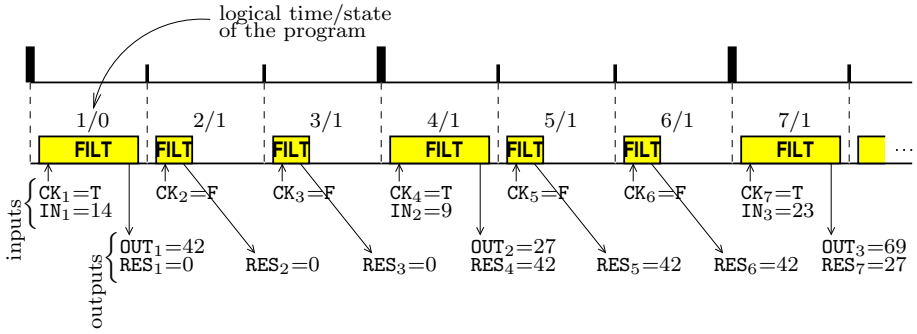

Fig. 4. A run of the centralized FILTER program.

The logical time of the program, mentioned in Figure 4, is the cycle number of the program, also called the base rate. Each other rate also defines a logical time, which is the logical time of all the variables with this rate; it is slower than the logical time of the program. In order to identify the values of the different variables, we add as a subscript the cycle number of its rate's logical time. For instance, CK_1=T, CK_2=F... Similarly, IN_1=14, IN_2=9... Note that the variables IN and RES advance at a different rate: the value IN_2 occurs at the same program cycle as the value RES_4. Note also that RES_4 is equal to OUT_1. Actually, the output RES is a "delayed" version of the output OUT. In this sense, the input value IN_1=14 yields the output value RES_4=42.

Throughout the paper, we suppose that SLOW takes 7 time units to complete and that, altogether, the other computations of the program take 1 time unit. Thus, the Worst Case Execution Time (WCET) of FILTER is 8 time units (7+1). In the run of Figure 4, this WCET is reached at cycles 1, 4, and 7. So the period of the execution loop (that is, the base rate of the program) cannot be smaller than 8 time units. This is not acceptable if the temporal requirements impose a smaller period. The purpose of our rate desynchronization method is to achieve a smaller period for the computations scheduled on the base rate. In Section 3, we show that the simple distribution of FILTER into two communicating tasks does not solve the problem

because these two tasks still share the same logical time, and hence they have the same period. Next, in Section 4, we present our method to further desynchronize the two tasks, so that the fast computations be performed at a fast rate, while the slow computations are performed at a slow rate.

3. A FIRST ATTEMPT OF DESYNCHRONIZATION

First, we present the basic distribution algorithm and the communication primitives used, before studying why it fails to work for the `FILTER` program.

3.1 Automatic distribution algorithm

The distribution algorithm we use in this paper is fully presented in [Caspi et al. 1999]. We only outline it here. It involves the following successive steps:

(1) assign a unique computing location to each sequential action, according to the *distribution specifications* provided by the user; these specifications are a partition of the set of inputs and outputs of the program into n subsets, one for each computing location of the final parallel program; note that achieving the "best" localization of the sequential actions (whatever the optimization criterion) is irrelevant in the present paper since we want to drive the distribution according to the rates of the program's inputs and outputs and not according to some optimization criterion; readers interested in these topics can refer to [Gupta et al. 1999];
(2) replicate the program on each location;
(3) on each location, suppress the sequential actions not belonging to the considered location;
(4) on each state of the automaton of each location, insert sending actions in order to solve the data dependencies between any two distinct locations;
(5) on each state of the automaton of each location, insert receiving actions in order to match the sending actions.

To make the obtained distributed programs less sensitive to communication latencies, the sends are inserted as early as possible in the OC program, while the receives are inserted as late as possible. Neither of the two insertion algorithms crosses the state frontier, that is, the send and receive actions are inserted state by state.

Finally, the correctness of our distribution algorithm has been formally proved in [Caillaud et al. 1997], meaning that the obtained distributed program is functionally equivalent to the initial centralized one.

3.2 Communication primitives

We choose to have two FIFO queues for each pair of locations, one in each direction. This is quite cheap in terms of execution environment, and has proved to work satisfactorily [Caspi and Girault 1995]. Concretely, we use two communication primitives:

—The *send* primitive `send(dst,var)` sends the current value of variable `var` to location `dst` by inserting it into the queue directed towards `dst`.

—The *receive* primitive `var:=receive(src)` extracts the head value from the queue starting at location `src` and assigns it to variable `var`, which is the local copy of the distant variable computed at location `src`.

These primitives perform both the data-transfer and the synchronization needed between locations: when the queue is empty, `receive` is blocking. The only requirement on the network is that it must preserve ordering and integrity of messages.

3.3 Distribution of the FILTER program

From now on, let us assume that the user wishes his `FILTER` program to run over two computing locations `L` and `M`, according to the following *rate distribution specifications*.

location name	assigned rates
L	base
M	CK

We now need to derive the corresponding distribution specifications, expressed on the inputs and outputs of the program instead of the assigned rates (as required by our distribution algorithm). To do this, we assign to a given computing location all the inputs and outputs whose rate belongs to it.

We also infer the rate of each location. Indeed, the knowledge of these rates will be required to compute the final WCET (in § 5.1). To compute the inferred rate of any given location, we take the root of the smallest subtree containing all the rates chosen by the user.

For the above distribution specifications, we obtain:

location name	assigned rates	inferred inputs & outputs	inferred location rate
L	base	CK, RES	base
M	CK	IN, OUT	CK

Applied to the `FILTER` program of Figure 2, these distribution specifications yield the following distributed program, shown in Figure 5.

location L (base rate)

state 0

```
go(CK);
send(M,CK);
if (CK) then {
  RES:=0;
  write(RES);
  goto 1;
} else {
  write(RES);
  goto 0;
} endif;
```

state 1

```
go(CK);
send(M,CK);
if (CK) then {
  OUT:=receive(M);
  RES:=OUT;
} else {
} endif;
write(RES);
goto 1;
```

location M (rate CK)

state 0

```
go(IN);
CK:=receive(L);
if (CK) then {
  OUT:=SLOW(IN);
  write(OUT);
  goto 1;
} else {
  goto 0;
} endif;
```

state 1

```
go(IN);
CK:=receive(L);
if (CK) then {
  send(L,OUT);
  OUT:=SLOW(IN);
  write(OUT);
} else {
} endif;
goto 1;
```

Fig. 5. The `FILTER` program distributed over two locations `L` and `M`.

The distributed program of Figure 5 calls for the following remarks:

(1) The `go(CK,IN)` action from the centralized `FILTER` program has been split into *two* actions: `go(CK)` on location `L` and `go(IN)` on location `M`. This is a direct consequence of our distribution specifications.

(2) On location `M`, all computations are scheduled on its rate `CK`. Hence, they are necessarily inside the `then` branch of the test `if(CK)`. Only a `send` or a `goto` can appear in the `else` branch, which is the case in state 0.

(3) The value of `CK` is sent by location `L` and received by location `M` *at each cycle* of the base rate. As a result, location `M` actually runs at the speed of the base rate instead of `CK`.

Figure 6 below is an example of a run of the distributed `FILTER` program, where `L` and `M` are both embedded in their own periodic execution loop.

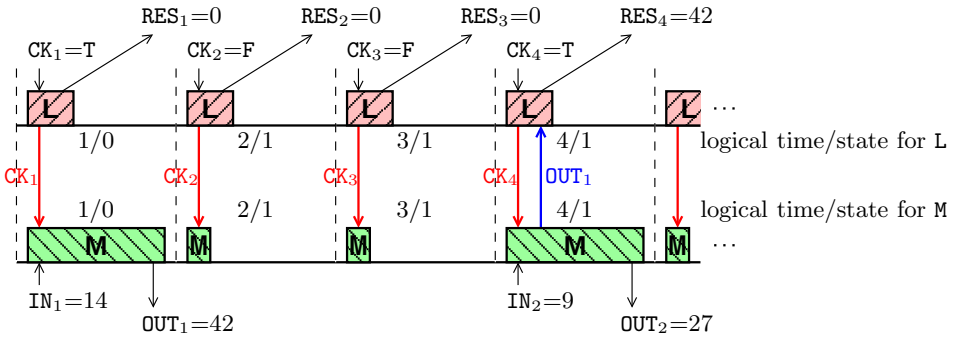

Fig. 6. A run of the distributed `FILTER` program.

Assume that the communications take 1 time unit. The new WCETs for `L` and `M` are respectively 2 and 8 time units. On a multiprocessor architecture, the global WCET is 8, hence the situation is the same as before; on a monoprocessor architecture, it is worse since the global WCET is 10. One reason is that `CK` is sent by `L` to `M` at each cycle. Would the period of `L` be smaller than that of `M`, the size of the FIFO queue would be unbounded.

Instead of the distributed program of Figure 5, we would like a program where location `M` runs at the speed of its assigned rate, that is `CK`. This would give enough time for the computation of the long duration task embedded in `SLOW`. In the next section, we explain in detail our method to achieve this.

4. THE PROPOSED METHOD

In location `M` of the distributed program of Figure 5, the `go(IN)` action could be performed *inside* the `then` branch of the test `if(CK)`. Indeed, the rate of `IN` is `CK`, and therefore a new value of `IN` is only expected at those cycles where `CK` is `true`. If we manage to move the `go(IN)` action inside the `then` branch of the test,

then all the actions performed by location `M` will be scheduled on `CK`, except the `CK:=receive(L)`. Hence, the idea is that a carefully chosen bisimulation should be able to detect that, in such a case, the `if(CK)` branching is *useless*. Suppressing this useless `if(CK)` test on location `M` will obviate the sending of `CK` by location `L`, and consequently the receiving of `CK` by location `M`, therefore allowing the two processes to run at different rates, that is, to have their rates *desynchronized*!

4.1 Moving the go actions downward

Moving the `go` actions downward concerns only the programs of the computing locations whose rates are *not* the base rate. For each such location, let `CK` be its rate and $\texttt{IN}_1,\ldots,\texttt{IN}_n$ be its n inputs, as specified by the distribution specifications. Hence, the current action at the root of its DAGs is $\texttt{go(IN}_1,\ldots,\texttt{IN}_n\texttt{)}$. We traverse each of its DAGs downward, starting from the root, as follows:

—On a unary node, continue in the next node.

—On a branching `if(var)`, if `var` is `CK`, then insert $\texttt{go(IN}_1,\ldots,\texttt{IN}_n\texttt{)}$ at the beginning of the `then` branch, mark the DAG, and terminate; otherwise continue in both branches `then` and `else`.

—On a branching closure, continue in the next node.

—On a leaf, do nothing.

—At the end of the traversal, if the DAG is marked, then remove the `go` at its root. It means that a `go` action has been inserted somewhere in the DAG.

To illustrate this algorithm, we apply it to the `FILTER` program. According to our distribution specifications (see § 3.3), only the DAGs of location `M` need to be traversed. Figure 7 shows the result for state 0 while Figure 8 shows the result for state 1.

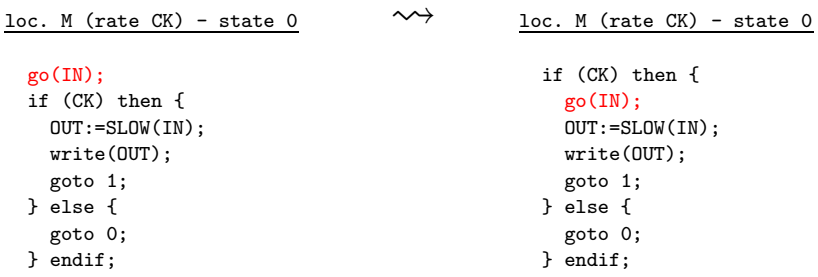

Fig. 7. State 0 of location `M`: (left) before and (right) after the traversal.

```
loc. M (rate CK) - state 1
```

```
go(IN);
if (CK) then {
  send(L,OUT);
  OUT:=SLOW(IN);
  write(OUT);
} else {
} endif;
goto 1;
```

$\rightsquigarrow$

```
loc. M (rate CK) - state 1
```

```
if (CK) then {
  go(IN);
  send(L,OUT);
  OUT:=SLOW(IN);
  write(OUT);
} else {
} endif;
goto 1;
```

Fig. 8. State 1 of location `M`: (left) before and (right) after the traversal.

4.2 Suppressing useless branchings

In [Caspi et al. 1995], we have presented an algorithm for suppressing binary branching whose branches are observationally equivalent. It involves traversing each automaton, starting from the first node of its initial state's DAG, and for each branching encountered, checking whether or not both branches are observationally equivalent. If they are, then the branching is suppressed; otherwise, it is kept. Checking if two branches are observationally equivalent is done on-the-fly with a bisimulation, called *test bisimulation*.

For our method to work, we split Step 4 of our distribution algorithm (see § 3.1) into two phases. The branching suppression will take place between these two phases. During the first phase, we only insert the `send` actions concerning the unary nodes of the DAG, that is, everything but the branchings. During the second phase, we insert the `send` actions concerning the binary nodes, that is, the branchings that *were not* suppressed.

The test bisimulation is formally expressed by three axioms and six inference rules (defined below). These axioms and rules apply to OC programs represented as CCS terms [Milner 1980] of the form:

$$q \ ::= \ nil \mid x \mid a.q \mid c_1.q + c_0.q \mid rec\, x.q$$

where x belongs to a finite set of variables. Intuitively, $a.q$ represents a unary node with action a, while $c_1.q + c_0.q$ represents a binary node with action c_1 leading to the `then` branch and action c_0 leading to the `else` branch, and finally $rec\, x.q$ is the starting point of a loop. Here a, c_1, and c_0 are abstract actions defining the concrete actions performed by the program.

We consider only *closed* and *well guarded* terms, that is without free variables (i.e., each variable instance is bound by a *rec*), and such that each variable occurs within some subterm $a.q'$, $c_1.q'$, or $c_0.q'$ (i.e., there is no empty loop such as $recx.x$). We call them *deterministic binary terms*, noted DBTs.

We also define *invisible* binary terms, noted IBTs, as closed and well-guarded terms of the form:

$$q \ ::= \ nil \mid x \mid c_1.q + c_0.q \mid rec\, x.q$$

The difference between an IBT and a DBT is that the only concrete actions performed by an IBT are binary branchings (i.e., binary DAG nodes), while a DBT performs both regular actions (i.e., unary DAG nodes) and binary branchings.

Indeed, for our purpose of branching suppression, we consider branchings to be *invisible* actions in the sense of observational equivalence. Note that any IBT is also a DBT.

For instance, to build the DBT of the `FILTER` program (location `M`), we start by defining the abstract actions according to Table I. Note that `goto` actions do not appear in the DBTs since we want to represent full programs and not just DAGs; hence `goto` actions are "expanded".

abstract action	concrete action
a	`go(IN)`
b	`send(L,OUT)`
d	`OUT:=SLOW(IN)`

abstract action	concrete action
e	`write(OUT)`
c_1	`if (CK) then`
c_0	`if (CK) else`

Table I. Abstract actions of the `FILTER` program (location `M`).

According to Table I, the DBT of the sole state 1, shown in Figure 8(right), is the subterm $p = [c_1.a.b.d.e.p + c_0.p]$. Introducing the *rec* notation gives the subterm $p = rec\ x.[c1.a.b.d.e.x + c_0.x]$. The DBT of the sole state 0, shown in Figure 7(right), is the subterm $q = [c_1.a.d.e.p + c_0.q]$. Again, introducing the *rec* notation and replacing p by the corresponding subterm for the full program gives the DBT $q = rec\ y.[c_1.a.d.e.rec\ x.[c_1.a.b.d.e.x + c_0.x] + c_0.y]$.

- axiom 1: $H \overset{v}{\vdash} \langle p, p\rangle$
- axiom 2: $H \overset{v}{\vdash} \langle s, t\rangle$
- axiom 3: $H \cup \langle p, q\rangle \overset{i}{\vdash} \langle p, q\rangle$
- rule 1: $\dfrac{H \overset{v}{\vdash} \langle p, q\rangle}{H \overset{i}{\vdash} \langle p, q\rangle}$
- rule 2: $\dfrac{H \overset{i}{\vdash} \langle p, q\rangle}{H \overset{v}{\vdash} \langle a.p, a.q\rangle}$
- rule 3: $\dfrac{H \overset{i}{\vdash} \langle p_1, q_1\rangle\ ,\ H \overset{i}{\vdash} \langle p_0, q_0\rangle}{H \overset{v}{\vdash} \langle c_1.p_1 + c_0.p_0, c_1.q_1 + c_0.q_0\rangle}$
- rule 4: $\dfrac{H \overset{v}{\vdash} \langle p_1, q\rangle\ ,\ H \overset{i}{\vdash} \langle p_0, q\rangle}{H \overset{v}{\vdash} \langle c_1.p_1 + c_0.p_0, q\rangle}$ and the three symmetrical rules
- rule 5: $\dfrac{H \overset{i}{\vdash} \langle p_1, q\rangle\ ,\ H \overset{i}{\vdash} \langle p_0, q\rangle}{H \overset{i}{\vdash} \langle c_1.p_1 + c_0.p_0, q\rangle}$ and the symmetrical rule
- rule 6: $\dfrac{H \cup \langle rec\ x.p, q\rangle \overset{v}{\vdash} \langle p[rec\ x.p/x], q\rangle}{H \overset{v}{\vdash} \langle rec\ x.p, q\rangle}$ and the symmetrical rule

Table II. Axioms and inference rules of the test bisimulation.

The axioms and inference rules formally defining our test bisimulation are presented in Table II. Here, p, q, p_0, q_0, p_1, and q_1 denote DBTs (or IBTs since an IBT is a DBT), while s and t denote only IBTs. Moreover, H is a set of pairs of terms,

with the meaning that $H \overset{i}{\vdash} \langle p, q\rangle$ or $H \overset{v}{\vdash} \langle p, q\rangle$ iff p and q are test bisimilar provided that each element of H is itself a pair of test bisimilar terms.

The principle is the following: for any pair of terms $\langle p, q\rangle$, if we can build a proof tree with root $\emptyset \overset{v}{\vdash} \langle p, q\rangle$, then p and q are test bisimilar, which we note $p \sim q$. The labels "i" and "v" allow the correct comparison of IBTs with DBTs. More specifically:

—$H \overset{i}{\vdash} \langle p, q\rangle$ is the root of a proof subtree whose nodes are exclusively instances of rule 5 and whose leaves are exclusively instances of axiom 3; this means that one of the two subterms only performs branchings, hence "i" for invisible;

—$H \overset{v}{\vdash} \langle p, q\rangle$ is the root of a proof subtree with at least one internal node being an instance of rule 2, 3, 4 or 6, or one leaf being an instance of axiom 1 or 2; this means that both subterms at least perform one identical action, hence "v" for visible.

The purpose of the three axioms and six rules is as follows:

—Axiom 1 states that a DBT is always test bisimilar to itself.

—Axiom 2 states that two IBTs are always test bisimilar.

—Axiom 3 and rule 6 allow any pair $\langle p, q\rangle$ to be considered *at most once*, even in the presence of loops in the DBTs. Indeed, loops in the DBTs could produce infinite branches in the proof tree of the test bisimulation. Instead, rule 6 adds the roots of such branches in the set of hypotheses H; these roots are either of the form $\langle rec\, x.p, q\rangle$ or $\langle p, rec\, x.q\rangle$. Then, axiom 3 allows us to cut these branches. By construction, $H \overset{v}{\vdash} \langle p, q\rangle$ means that $p \sim q$ under the hypotheses that all elements of H are pairs of test bisimilar terms.

—Rule 1 says that the $\overset{v}{\vdash}$ relation is stronger than the $\overset{i}{\vdash}$ relation.

—Rule 2 says that if p and q are test bisimilar, then so are $a.p$ and $a.q$.

—Rule 3 says that if p_1 and q_1 are test bisimilar, as well as p_0 and q_0, then so are $c_1.p_1 + c_0.p_0$ and $c_1.q_1 + c_0.q_0$.

—Rule 4 says that if p_1 and q are test bisimilar, as well as p_0 and q, and if additionally at least one pair is test bisimilar under the $\overset{v}{\vdash}$ relation, then so are $c_1.p_1 + c_0.p_0$ and q. The three symmetrical rules express all the alternatives regarding the "v" and "i" labels on the rule's premise, and regarding the left and right terms of the rule's conclusion.

—Rule 5 is weaker than rule 4 in the sense that no $\overset{v}{\vdash}$ is required for the pairs $\langle p_1, q\rangle$ and $\langle p_0, q\rangle$, but that the pair $\langle c_1.p_1 + c_0.p_0, q\rangle$ is test bisimilar under the $\overset{i}{\vdash}$

relation instead of $\overset{v}{\vdash}$. The symmetrical rule expresses the alternative regarding the left and right terms.

We have proved in [Caspi et al. 1995] that the inference system of Table II is sound and complete. In other words:

$$\emptyset \overset{v}{\vdash} \langle p, q\rangle \iff p \sim q$$

Actually building the proof tree with a reasonable time and space complexity is difficult. For this purpose, we use an on-the-fly algorithm inspired from [Fernandez and Mounier 1991].

We illustrate our test bisimulation with our `FILTER` program (location `M`). For the sake of brevity, we consider here only the DBT of state 1 (shown in Figure 8(right)), which is $p = rec\, x.[c_1.a.b.d.e.x + c_0.x]$. We first traverse the DAG until we reach the branching. At this point, we call the test bisimulation checker with the initial pair $\langle a.b.d.e.rec\, x.[c_1.a.b.d.e.x + c_0.x], rec\, x.[c_1.a.b.d.e.x + c_0.x]\rangle = \langle a.b.d.e.p, p\rangle$ (for convenience, we write p instead of $rec\, x.[c_1.a.b.d.e.x + c_0.x]$). We thus build the proof tree of Figure 9.

$$\dfrac{\text{rule 6}\quad \dfrac{\text{rule 4}\quad \dfrac{\text{axiom 1}\quad \overline{\{\langle a.b.d.e.p, p\rangle\} \overset{v}{\vdash} \langle a.b.d.e.p, a.b.d.e.p\rangle} \qquad \overline{\{\langle a.b.d.e.p, p\rangle\} \overset{i}{\vdash} \langle a.b.d.e.p, p\rangle}\ \text{axiom 3}}{\{\langle a.b.d.e.p, p\rangle\} \overset{v}{\vdash} \langle a.b.d.e.p, c_1.a.b.d.e.p + c_0.p\rangle}}{\emptyset \overset{v}{\vdash} \langle a.b.d.e.p, p\rangle}$$

Fig. 9. Proof tree for the DBT of program `FILTER`'s location `M`.

Building the proof tree of Figure 9 proceeds in three steps. The first step uses rule 6 to unroll the loop once: $p = rec x.[c_1.a.b.d.e.x + c_0.x]$ is thus transformed into $c_1.a.b.d.e.rec x.[c_1.a.b.d.e.x + c_0.x] + c_0.rec x.[c_1.a.b.d.e.x + c_0.x] = c_1.a.b.d.e.p + c_0.p$ and the pair $\langle a.b.d.e.p, p\rangle$ is added to the empty hypotheses set. In the second step, rule 4 consumes the branching c_1/c_0 and creates two branches in our proof tree. The left branch is ended thanks to axiom 1. The right branch is ended thanks to axiom 3. As a result, $a.b.d.e.p \sim p$, and the branching can be suppressed. Therefore, the DBT for Figure 8(right) is reduced to $p' = rec\, x.a.b.d.e.x$.

Figure 10 shows the result of applying our test bisimulation to the DAGs of location `M`. In state 1, the parts labeled q_1 and q_0 are found to be bisimilar, hence the branching is suppressed (see the proof tree above). The same goes in state 0 for p_1 and p_0, even though there is a `goto` in the `else` branch.

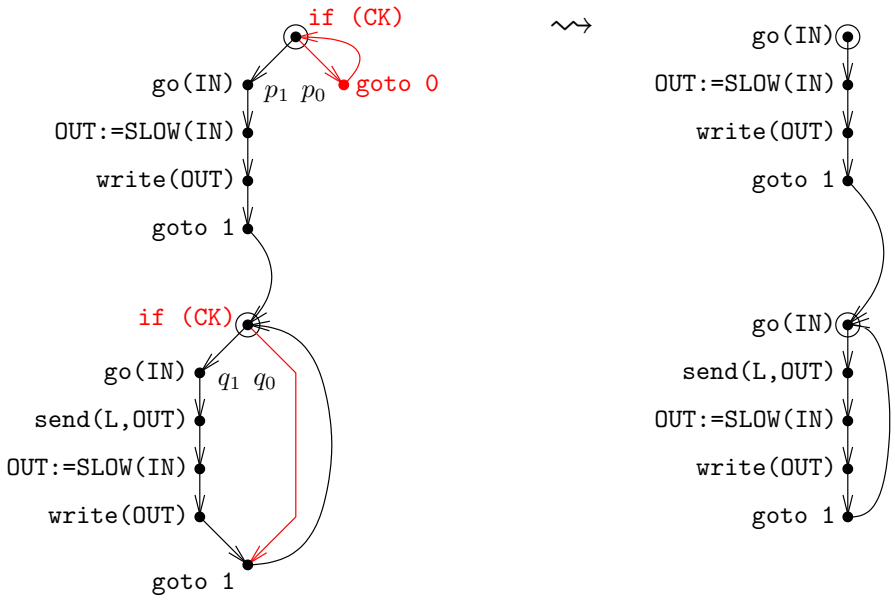

Fig. 10. Suppression of the branchings in the program of location M.

4.3 Discussion

At this point, there remains to insert the **send** actions for the condition variables of the branchings that have not been suppressed. For the FILTER program, the only remaining branching is the if(CK) of location L, and since CK belongs to location L, nothing is inserted. Finally, all the **receive** actions need to be inserted. For the FILTER program, the corresponding result is shown in Figure 11:

location L (base rate)

state 0

```
go(CK);
if (CK) then {
  RES:=0;
  write(RES);
  goto 1;
} else {
  write(RES);
  goto 0;
} endif;
```

state 1

```
go(CK);
if (CK) then {
  OUT:=receive(M);
  RES:=OUT;
} else {
} endif;
write(RES);
goto 1;
```

location M (rate CK)

state 0

```
go(IN);
OUT:=SLOW(IN);
write(OUT);
goto 1;
```

state 1

```
go(IN);
send(L,OUT);
OUT:=SLOW(IN);
write(OUT);
goto 1;
```

Fig. 11. The final FILTER program distributed over two locations L and M.

Figure 12 shows a run of the new rate desynchronized FILTER program on a distributed architecture, to be compared with Figure 6.

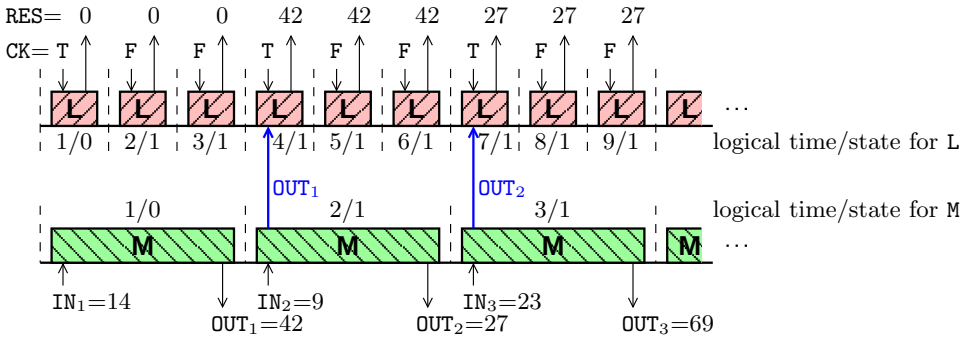

Fig. 12. A run of the rate desynchronized `FILTER` program.

Observe that the period of `L` can now be smaller than that of `M`. In this run, the former is one third of the latter, `CK` being `true` once every three instants. For instance, with the same durations as in § 3.3, the periods of `L` and `M` can be respectively 3 and 9.

On a single processor architecture with a preemptive operating system and a RM scheduling policy, Figure 13 shows the schedule obtained when the periods of `L` and `M` are respectively 5 and 15. No communication takes place before the second occurrence of `CK`, which corresponds to cycle 4 of task `L` and to the beginning of cycle 2 of task `M`. At this point, task `L` begins and is suspended as soon as it reaches the `OUT:=receive(M)` action, since the queue is empty. Then, `M` begins and immediately sends `OUT`, which resumes `L`. Since `L` has a higher priority, it preempts `M` and completes its execution. Then, `M` resumes and continues until the beginning of `L`'s cycle 5 when it is preempted by `L`. This results in slicing `M` into `M1`, `M2`, and `M3`. Here, the deadlines are met thanks to the distribution into two rate desynchronized processes.

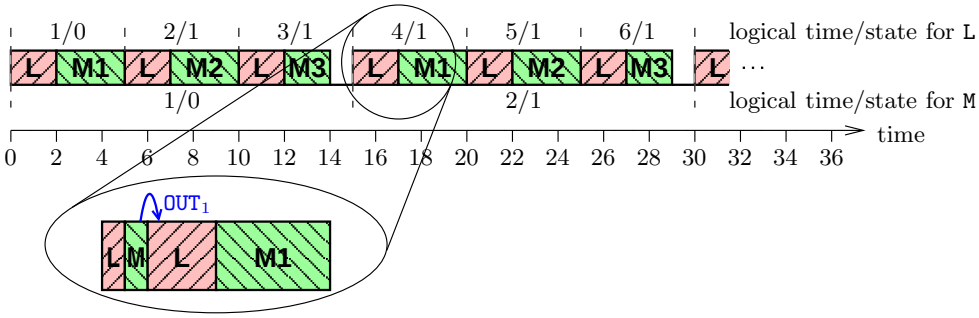

Fig. 13. Centralized schedule obtained with the rate desynchronization.

In the next section, we present a data-flow analysis that will move the `send` actions backward in order to prevent this unnecessary task switching.

4.4 Data-flow analysis

Figure 13 shows that a lot of unnecessary task switching may occur. We propose to apply a tailored data-flow analysis to move the sending of OUT from M to L backward, that is just after the assignment to OUT. As a result, the preemptions taking place in Figure 13 will not occur, therefore resulting in the run shown in Figure 14.

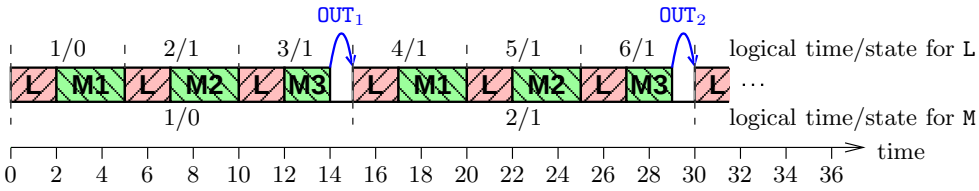

Fig. 14. Centralized schedule obtained when OUT is sent as soon as possible.

The goal of this data-flow analysis is to move the send actions backward. As explained in § 3.1, the send actions are initially inserted on a state by state basis (step 4). In contrast, here, we do cross the state frontier. Also, this analysis takes place *before* the insertion of the receive actions, to avoid messing with the order of the values transmitted in the FIFO queues.

For this purpose, we define for any action a the set def(a) of the variables it defines (e.g., {x} for an assignment x:=e or a go(x)).

Then, for each location s, for each DAG of this location's program, we start a traversal at each leaf of this DAG. For each other location t, we define one set $\texttt{Sent}_{\texttt{s,t}}$, initially empty, which will contain at any point the variables sent by s to t; we propagate these sets upward in the following way:[4]

(1) when reaching a send(t,x), remove it from the DAG and insert x in $\texttt{Sent}_{\texttt{s,t}}$;

(2) when reaching an action a, for each $\texttt{x} \in \texttt{def(a)}$ and for each t such that $\texttt{x} \in \texttt{Sent}_{\texttt{s,t}}$, remove x from $\texttt{Sent}_{\texttt{s,t}}$ and insert a send(t,x) in the DAG just after the action a;

(3) when reaching a branching closure, duplicate each set $\texttt{Sent}_{\texttt{s,t}}$ and proceed in both branches then and else;

(4) when reaching a branching, compute each intersection $\texttt{Sent}^{\texttt{inter}}_{\texttt{s,t}}$ of the sets $\texttt{Sent}^{\texttt{then}}_{\texttt{s,t}}$ and $\texttt{Sent}^{\texttt{else}}_{\texttt{s,t}}$; in the then branch, for each variable x in $\texttt{Sent}^{\texttt{then}}_{\texttt{s,t}} - \texttt{Sent}^{\texttt{inter}}_{\texttt{s,t}}$, insert a send(t,x); do similarly in the else branch; proceed in the DAG above the branching with the set $\texttt{Sent}^{\texttt{inter}}_{\texttt{s,t}}$;

[4]We process one location at a time, but actually our software has a unique data structure for all the locations, consisting of an ordered list of DAGs, where each node is labeled with the set of locations that must compute its action. Hence, instead of performing n traversals, one for each location s, we only perform *one* traversal.

(5) when reaching the root of a DAG that is not the program's initial state, say state `u`, replicate each non empty set $\texttt{Sent}_{s,t}$ into as many copies as leaves `goto u` that exist in the program, and proceed in all theses leaves; for all the other empty sets $\texttt{Sent}_{s,t}$, the traversal terminates here.

This traversal terminates because the number of states is finite, and because any sent value is always expected by the destination location in the same DAG where the `send` action appears initially (this is true by construction, see [Caspi et al. 1999]). Hence, any variable inserted in a $\texttt{Sent}_{s,t}$ set is extracted from this set either during the first traversal of the DAG where it was inserted (if the action that needs its value is "above" the initial `send`), or during the second traversal (if it is "below"). As a result, in both cases the set $\texttt{Sent}_{s,t}$ is empty after the second traversal of this DAG when it reaches the root, so the algorithm terminates.

Concerning our example, the program of location `L` does not contain any `send`, so this data-flow analysis leaves it unmodified. The program of location `M` contains two leaves, respectively in states 0 and 1. The DAG of state 0 does not contain any `send`, so the traversal initiated at the leaf of this DAG has no effect. The DAG of state 1 contains one `send(L,OUT)`, just below the `go(IN)`. Here is what happens: the traversal is initiated at the leaf `goto 1` with an empty set $\texttt{Sent}_{M,L}$. When the `send(L,OUT` is reached, it is removed from the DAG and `OUT` is inserted in $\texttt{Sent}_{M,L}$. Then, the root of state 1 is reached (`go(IN)`), which has two incoming transitions, one from state 0 and one from state 1. The traversal is thus resumed from the two corresponding leaves `goto 1`. In the DAG of state 0, the `write(OUT)` is first reached, with no effect. Then, the `OUT:=SLOW(IN)` is reached: since $\texttt{OUT} \in \texttt{Sent}_{M,L}$, a `send(L,OUT)` is inserted at this point and $\texttt{Sent}_{M,L}$ becomes empty. Exactly the same thing happens in the DAG of state 1. This traversal is illustrated in Figure 15, where the `send(L,OUT)` in the dashed box is removed and replaced by the two `send(L,OUT)` in the solid boxes.

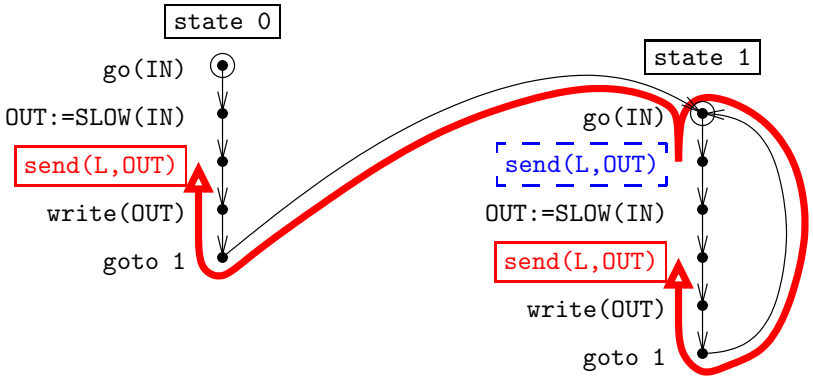

Fig. 15. Traversal of the `FILTER` program, location `M`.

4.5 Inserting the `receive` actions

What remains to do is to insert the `receive` actions. The algorithm of § 3.1-step 5 works, but on a state by state basis. For the same reason advocated when moving

the `send` actions backward, we do want to cross the state frontier. Therefore, we propose a new `receive` insertion algorithm. For this purpose, we define, for any action `a`, the set `use(a)` of the variables it uses (e.g., `{y,z}` for the assignment `x:=y+z`). Our algorithm consists of a traversal of the program, starting from the root of the initial state's DAG, with a FIFO queue $\texttt{Queue}_{\texttt{s,t}}$ for each pair of locations (`s`,`t`), which will contain at any point the variables sent by `s` to `t`, in their sending order (hence the usage of FIFO queues instead of sets). These initially empty FIFO queues are propagated forward in the following way:[5]

(1) when reaching a `send(t,x)` computed by `s`, insert `x` at the tail of $\texttt{Queue}_{\texttt{s,t}}$;

(2) when reaching a leaf `goto u`, propagate all the queues $\texttt{Queue}_{\texttt{s,t}}$ to the root of the DAG of state `u`, and resume the traversal at the root of this DAG;

(3) when reaching a DAG's root, if it is unmarked, mark it and propagate all the queues $\texttt{Queue}_{\texttt{s,t}}$ to the first node of the DAG; if it is already marked, do nothing;

(4) when reaching an action `a` computed by `t`, for each $\texttt{x} \in \texttt{use(a)}$ computed by `s` where $\texttt{s} \neq \texttt{t}$ (hence `x` is a distant variable of `t`), if $\texttt{x} \in \texttt{Queue}_{\texttt{s,t}}$, then extract the head variable `h` from $\texttt{Queue}_{\texttt{s,t}}$ and insert a `h:=receive(s)` action in the DAG of location `t`, until the variable `x` is extracted; this insures that the `receive` actions are inserted in location `t` in an order that matches exactly that of the `send` actions in location `s`;

(5) when reaching a branching, duplicate each queue $\texttt{Queue}_{\texttt{s,t}}$ and proceed in both branches `then` and `else`;

(6) when reaching a branching closure, for each pair of locations (`s`,`t`), compute the greatest common suffix $\texttt{Queue}^{\texttt{suffix}}_{\texttt{s,t}}$ of the queues $\texttt{Queue}^{\texttt{then}}_{\texttt{s,t}}$ and $\texttt{Queue}^{\texttt{else}}_{\texttt{s,t}}$; it contains the variables that have been sent by `s` to `t` in both branches and such that the sends are the closest to the branching closure; they can therefore be received after the closure; so, in the `then` branch, for each location `t`, extract each variable `h` from the head of the truncated queue $\texttt{Queue}^{\texttt{then}}_{\texttt{s,t}} - \texttt{Queue}^{\texttt{suffix}}_{\texttt{s,t}}$, and insert a `h:=receive(s)` action in the DAG of location `t`, until the queue is empty; do similarly in the `else` branch; proceed in the DAG below the branching closure with the common suffix $\texttt{Queue}^{\texttt{suffix}}_{\texttt{s,t}}$.

This traversal terminates when all the DAGs are marked. Moreover, the traversal is consistent because, when a root is pointed to by several leaves (i.e., several `goto u`), then the queue $\texttt{Queue}_{\texttt{s,t}}$ that will be propagated from each of these leaves will contain exactly the same variables in the same order.

Concerning our example, since there is no `send` from `L` to `M`, the queues $\texttt{Queue}_{\texttt{L,M}}$ remain empty during the whole traversal, so nothing happens for `M`.

[5]Remember that the data structure handled by our software consists of an ordered list of DAGs, where each node is labeled with the set of locations that must compute its action.

Figure 16 illustrates what happens for L. As said above (footnote 5), the data structure traversed by the receive insertion algorithm consists of a single ordered list of DAGs, where each node is labeled with the set of locations that must compute its action. For instance, the unary node "(L) RES:=0" means that the action RES:=0 must be computed by the sole location L. Hence, this single data structure represents the program of both locations L and M. Here, only the queue $\texttt{Queue}_{\texttt{M,L}}$ is depicted: "∅" means that it is empty, while "x,y]" means that it contains the values of x and y, with y being the head value.

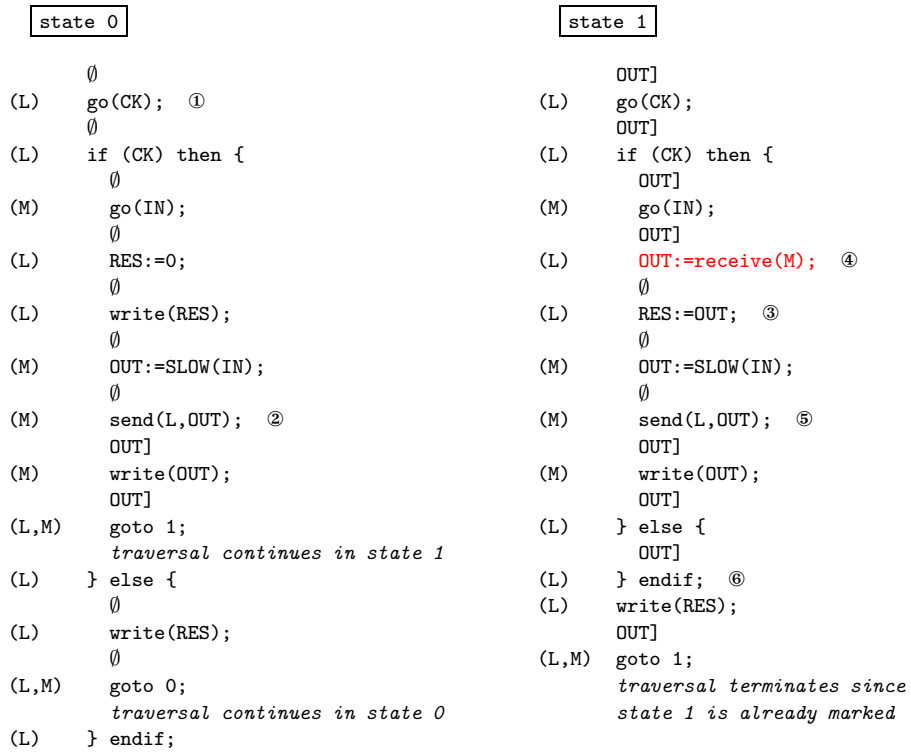

Fig. 16. Insertion of the receive actions on the rate desynchronized FILTER program.

The traversal begins at the root of state 0's DAG, with the queue $\texttt{Queue}_{\texttt{M,L}}$ being initially empty (at point ①). When traversing the DAG of state 0, in the then branch a send(L,OUT) is encountered (at point ②), so OUT is inserted at the tail of $\texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$. Since none of the actions encountered after has OUT in its use set, the leaf goto 1 is reached with still $\texttt{OUT} \in \texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$. In contrast, in the else branch, $\texttt{Queue}^{\texttt{else}}_{\texttt{M,L}}$ is empty when the leaf goto 0 is reached. Note that the branching closure is never reached because of the two leaves.

As a consequence, the traversal resumes in state 1 with $\texttt{Queue}_{\texttt{M,L}}$ containing OUT alone. In the then branch, the action RES:=OUT is encountered, with OUT in its use set (at point ③); OUT being in $\texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$, it is extracted from the head of $\texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$

and the action `OUT:=receive(M)` is inserted in the DAG of location `L` (at point ④). The queue $\texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$ is then empty. A second `send(L,OUT)` is encountered (at point ⑤), so `OUT` is once again inserted at the tail of $\texttt{Queue}^{\texttt{then}}_{\texttt{M,L}}$, and the branching closure is reached (at point ⑥). In the `else` branch, none of the actions encountered has `OUT` in its `use` set, so the branching closure is reached with still $\texttt{OUT} \in \texttt{Queue}^{\texttt{else}}_{\texttt{M,L}}$ (also at point ⑥). Hence, the greatest common suffix contains only `OUT`, while both truncated queues are empty. Therefore, no `receive` action is inserted in the `then` and `else` branches, and the leaf `goto 1` is reached with $\texttt{OUT} \in \texttt{Queue}_{\texttt{M,L}}$. The traversal terminates here since the DAG of state 1 is already marked.

As a result, the final programs of locations `L` and `M` are shown in Figure 17. In location `M`, the codes of states 0 and 1 are identical. An automaton minimization tool can detect this and remove state 1, replacing the `goto 1` in state 0 by a `goto 0`.

location L (base rate)

state 0

```
go(CK);
if (CK) then {
  RES:=0;
  write(RES);
  goto 1;
} else {
  write(RES);
  goto 0;
} endif;
```

state 1

```
go(CK);
if (CK) then {
  OUT:=receive(M);
  RES:=OUT;
} else {
} endif;
write(RES);
goto 1;
```

location M (rate CK)

state 0

```
go(IN);
OUT:=SLOW(IN);
send(L,OUT);
write(OUT);
goto 1;
```

state 1

```
go(IN);
OUT:=SLOW(IN);
send(L,OUT);
write(OUT);
goto 1;
```

Fig. 17. The rate desynchronized `FILTER` program after the data-flow analysis.

4.6 The new distribution algorithm

Taking into account the modifications described in § 4.1 and 4.4, our new rate desynchronization algorithm involves the following successive steps:

(1) locate each sequential action; same as § 3.1-Step 1;
(2) replicate the program; same as § 3.1-Step 2;
(3) suppress the sequential actions; same as § 3.1-Step 3;
(4) on each state and each location, move the `go` action downward according to the rate of the computing location; see § 4.1;
(5) on each state of the automaton, insert `send` actions to solve the data dependencies between two distinct locations, *except* when concerning branching actions;
(6) on each state and each location, suppress the useless branchings; see § 4.2;
(7) on each state, insert `send` actions to solve the data dependencies concerning the branching actions that were not suppressed during Step 6;
(8) move the `send` actions upward with a data-flow analysis; see § 4.4;
(9) insert the `receive` actions with the new algorithm; see § 4.5.

5. DISCUSSION

5.1 Checking the temporal requirements

Checking the temporal requirements is always crucial when programming embedded reactive systems. This involves computing the WCET of the object code generated by the compiler, and comparing it to the execution loop period it is embedded in (see § 1.2). In our case, it is more complex because the program is distributed into n processes. Each of the n processes is an OC program, a finite state automaton with a DAG of actions in each state. We compute the WCET of an OC program as the maximum of its DAGs' WCETs. This is where the long duration tasks are taken into account. We thus have n WCET, each corresponding to a program whose execution rate is the location's rate (see § 3.3).

—If each location's rate is *constant* (i.e., does not vary dynamically), then we can compute the utilization factor of each processor and check the Liu & Layland condition to know whether a static or dynamic priority schedule is feasible or not [Liu and Layland 1973]: namely Rate Monotonic (RM) for the static priority policy and Earliest Deadline First (EDF) for the dynamic priority policy. Note that to avoid uncontrollable context switching between tasks, static priorities are preferable for hard real-time systems.[6]

Concerning our `FILTER` example, the WCET of locations `L` and `M` are respectively 2 and 8. Since the rates of locations `L` and `M` are respectively 5 and 15, the Liu & Layland condition for RM holds:[7]

$$\frac{2}{5} + \frac{8}{15} = \frac{14}{15} \leq 1$$

As a result, the schedule shown in Figure 13 above is feasible with static priorities under the RM policy.

—Now, what happens if some rates vary dynamically? For instance, a given input can be present each time some other input is greater than some threshold. In such a case, the Liu & Layland conditions are useless, so checking the temporal requirements is much more difficult. Yet, it can be the case that each rate follows a cyclic pattern, with the WCET of each program following the same cyclic pattern (otherwise, as explained in Section 1.3, there is an inconsistency). For instance, a given input can be present twice every three iterations. It is then possible to check the Liu & Layland conditions for each combination of presence and absence of the inputs and outputs, the total number of such combinations being the meta-period of the cycles.

5.2 Implementation

The method presented in this paper has been implemented in the `ocrep` tool[8], to distribute automatically OC programs with the bisimulation-based branching

[6] Although this point is still being debated by the real-time scheduling community.
[7] 15 being a multiple of 5, the bound on the utilization factor is 1 instead of $2(\sqrt{2}-1)$.
[8] `ocrep` is available at `http://pop-art.inrialpes.fr/~girault/Ocrep`.

reduction.

It has been successfully tested on several OC programs produced by either the LUSTRE [Halbwachs et al. 1991] or the ESTEREL [Berry and Gonthier 1992] compilers (a two player tennis game, a digital wristwatch, and various control software for car making factories). Both are synchronous programming languages, particularly well suited to designing reactive embedded systems [Benveniste et al. 2003]. The synchronous approach has been proposed to *ease* the design of reactive systems. It is based on the so-called *synchronous abstraction*. Without entering into details, it is similar to the abstraction made when designing synchronous circuits at the gate level. The `ocrep` tool acts as a post-processor for the LUSTRE and ESTEREL compilers. Concretely, it takes as input an OC program `foo.oc`, and a file `foo.rep` containing the distribution specifications (see item 1 in § 3.1).

In the following two sections, we give details of how our method is applied to LUSTRE and ESTEREL programs.

5.3 Automatic clock-driven distribution of LUSTRE programs

LUSTRE is a data-flow synchronous programming language [Halbwachs et al. 1991]. Like SIGNAL/ POLYCHRONY [Le Guernic et al. 1991] and LUCID Synchrone [Caspi and Pouzet 1996], it use *clocks* as powerful control structures to manipulate data [Caspi 1992; Colaço and Pouzet 2003]. In data-flow languages, each variable manipulated by the program is a *stream*, which is an infinite sequence of typed data, and clocks are a form of *temporal types*. The clock of a stream defines the sequence of logical instants where the stream bears a value. In LUSTRE, any Boolean stream can be a clock. A predefined clock always exists: it is the *base clock* of the program, which is the sequence of its activation instants. That is, the base clock is the stream of `tick`s of the execution loop. LUSTRE, SIGNAL/POLYCHRONY, and LUCID Synchrone offer operators to upsample and downsample streams. Downsampling allows the definition of a *slower* clock, while upsampling allows the projection of a slow stream onto a faster clock. All the clocks of a program can be represented within a single *tree of clocks*, whose root is the base clock.[9] A clock `C1` is then said to be *faster* than another clock `C2` if `C2` is in the subtree whose root is `C1`. Clocks look very close to the rates introduced in § 2.2. However, a clock has a more powerful meaning due to its language nature (i.e., it is a temporal type), something that a rate lacks.

```
node FILTER (CK : bool; (IN : int) when CK)
     returns (RES : int; (OUT : int) when CK);
let
  RES = current ((0 when CK) -> pre OUT);
  OUT = SLOW (IN);
tel;
function SLOW (A : int) returns (B : int);
```

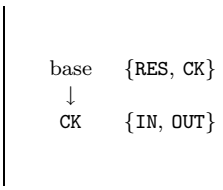

Fig. 18. The LUSTRE `FILTER` program and its clock tree.

Figure 18 shows an example of a LUSTRE program (to the left) along with its

[9] Actually, this is not true in SIGNAL/POLYCHRONY, where it is a *forest* instead of a *tree*.

clock tree (to the right). Note that `when`, `current`, `pre`, and `->` are respectively the downsampling, upsampling, delay, and initialization operators. Compiling this LUSTRE program gives the OC program of Figure 2. Each node of its clock tree is decorated with the inputs and outputs whose clock matches the node. Note that this clock tree is identical to the rate tree of the `FILTER` OC program.

The following table gives an example of a run of this program, corresponding to the `FILTER` OC program's run shown in Figure 4:

cycle number	1	2	3	4	5	6	7	8	9	...
CK	T	F	F	T	F	F	T	F	F	...
IN	14			9			23			...
OUT	42			27			69			...
pre OUT	nil			42			27			...
0 when CK	0			0			0			...
(0 when CK) -> pre OUT	0			42			27			...
RES	0	0	0	42	42	42	27	27	27	...

As we have already noted for Figure 4, the input value $\texttt{IN}_1$=14 yields the output value $\texttt{RES}_4$=42. The delay in cycles between these two values (the first cycle for `IN` versus the fourth cycle for `RES`) is due to the `pre` operator: it delays the upsampling of the `OUT` value of one cycle of the slow clock `CK`. In other words, it "gives time" to the `SLOW` computation to terminate before using its result to compute `RES`. As we have seen in § 4, the corresponding OC program obtained can be rate desynchronized. However, the same LUSTRE program without the `pre` cannot be rate desynchronized. The technical reason is that the test bisimulation does not suppress the `if(CK)` branching on location `M`. It makes sense because, without the `pre` operator, the result of the `SLOW` computation is used *at once* to produce the value of `RES`. It is as if we were trying to schedule a long duration task at a fast rate, which is of course impossible.

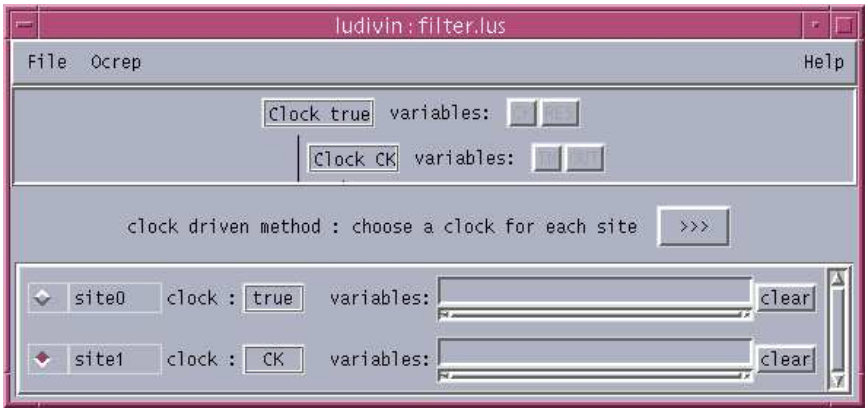

Fig. 19. A screen-shot of the `ludivin` GUI.

Finally, the `ludivin`[10] GUI [Salpétrier 2002] has been developed to build automatically the clock tree of a LUSTRE program, help the user specify a desired distribution, and call `ocrep` to perform the rate desynchronization. Figure 19 is a screen-shot of the `ludivin` GUI, opened with the `FILTER` program. To avoid name conflicts, the base clock is named `true`, which is a keyword of LUSTRE.

5.4 Automatic rate desynchronization of ESTEREL programs

ESTEREL is an imperative synchronous programming language [Berry and Gonthier 1992], now commercialized by ESTEREL TECHNOLOGIES, with successes in electronic design automation and avionics. It shares with LUSTRE the synchronous abstraction, and programs can be compiled into the same automaton format OC.

It is possible to obtain an OC program *similar* to the one of Figure 2 (not exactly the same but having the same overall behavior) from the ESTEREL compiler. The corresponding ESTEREL program is shown in Figure 20 below. The "`loop ... each E`" construct restarts its body each time the event `E` is present. A signal (input or output) is present if and only if it is emitted during the current reaction. The "`||`" construct launches its two branches simultaneously and terminates as soon as both branches terminate. The "`emit X (V)`" construct emits the output `X` with the value `V`, "`?X`" is the current value of `X`, and "`pre(X)`" is the previous value of `X`. The "`suspend ... when E`" construct suspends its body as soon as the event `E` is present. Finally, "`immediate`" implies that this suspension can occur in the same instant than the one when the body was started.

```
module FILTER:
  input CK;
  input IN : integer;
  output RES : integer;
  output OUT : integer;
  function SLOW(integer): integer;
  loop
      suspend
        emit OUT (SLOW(?IN))
      when immediate [not CK]
    ||
      emit RES (pre(?OUT))
  each tick
end module
```

Fig. 20. The ESTEREL source code for `FILTER`.

Compiling the above ESTEREL program produces the OC automaton of Figure 21. Compared to OC programs obtained from the LUSTRE compiler, there are three particularities: first, the `reset` action assigns `0` to the variable in argument; second, `_V_5` is a local variable used to implement the `pre` operator; and third, `_P_IN` is the Boolean encoding the presence information of the input `IN` (and similarly for `_P_CK`). For the sake of clarity and space, Figure 21 shows a *simplified* version of the

[10] `ludivin` is available at `http://pop-art.inrialpes.fr/~girault/Ludivin`.

OC program, where the `sink` state has been removed, and where state 2, identical to state 1, has also been removed.

`state 0`

```
go(CK,IN);
reset(RES);
reset(OUT);
reset(_V_5);
if (_P_CK) then {
  if (_P_IN) then {
  } else {
    reset(IN);
  }
  RES:=_V_5;
  write(RES);
  OUT:=SLOW(IN);
  write(OUT);
  _V_5:=OUT;
  goto 1;
} else {
} endif;
if (_P_IN) then {
} else {
  reset(IN);
}
RES:=_V_5;
write(RES);
goto 1;
```

`state 1`

```
go(CK,IN);
RES:=_V_5;
write(RES);
if (_P_CK) then {
  OUT:=SLOW(IN);
  write(OUT);
  _V_5:=OUT;
  goto 1;
} else {
}
goto 1;
```

Fig. 21. OC program obtained by compiling the `FILTER` ESTEREL program.

location `L` (base rate)

`state 0`

```
go(CK);
reset(RES);
reset(_V_5);
if (_P_CK) {
  RES:=_V_5;
  write(RES);
  OUT:=receive(M);
  _V_5:=OUT;
  goto 1;
} else {
}
RES:=_V_5;
write(RES);
goto 1;
```

`state 1`

```
go(CK);
RES:=_V_5;
write(RES);
if (_P_CK) then {
  OUT:=receive(M);
  _V_5:=OUT;
  goto 1;
} else {
} endif;
goto 1;
```

location `M` (rate `CK`)

`state 0`

```
go(IN);
reset(OUT);
if (_P_IN) {
} else {
  reset(IN);
}
OUT:=SLOW(IN);
send(L,OUT);
write(OUT);
goto 1;
```

`state 1`

```
go(IN);
OUT:=SLOW(IN);
send(L,OUT);
write(OUT);
goto 1;
```

Fig. 22. The rate desynchronized program obtained by distributing the OC program of Figure 21.

There is no clock in ESTEREL, so the clock-driven distribution method described for LUSTRE programs in § 5.3 cannot be used. However, our rate desynchronization method described in § 4 achieves the same result as with the OC `FILTER` program of Figure 3; that is, it produces a distributed program where the part computing the `SLOW` function (i.e., location `M`) can be run at the rate defined by the boolean input `CK`. The obtained rate desynchronized OC program is shown in Figure 22. Observe that, in location `M`, all the branchings `if (_P_CK)` have been suppressed.

When distributing LUSTRE programs, the success of the method was expected since LUSTRE clocks are very close to OC rates. In contrast, for ESTEREL programs, this is more surprising, because there is no clock in ESTEREL. In fact, our rate desynchronization method does manage to produce distributed programs that can be run at different rates, provided that the computations are sufficiently decoupled. In some sense, this means that we were able to "create" a clock in the ESTEREL program. Note that our method also "creates" clocks with LUSTRE programs where there is no clock but where some computations are also sufficiently decoupled.

5.5 Related work

There has been a lot of work in the research area of automatic distribution & parallelization in a general way. Reference [Gupta et al. 1999] is a good survey paper on this topic, with its Section 5 being devoted to distributed memory parallelism. Yet, none of the existing general methods tackles the specific problem of long duration tasks, rate desynchronization, or clock driven distribution. To the best of our knowledge, our method is the only one to address this specific problem. Nonetheless, there are a number of related approaches worth mentioning.

Our work is related to GIOTTO [Henzinger et al. 2003], a compiler for embedded systems. Its abstraction consists of instantaneous communication, time-deterministic computation, and value-deterministic computation. This is very much like the synchronous abstraction. GIOTTO's basic functional unit is the task, which is a periodically executed piece of code. One important point is that each period must be *static*. In contrast, rates (or clocks) can specify periods that change *dynamically*, even though the consistency checking between the clock (which is non periodic), the task execution time (which must also vary dynamically) and the processor performances is more complex. A GIOTTO program can also be annotated with *platform constraints*, which are similar to our distribution specifications: a constraint may map a particular task onto a particular CPU. The GIOTTO compiler schedules the tasks onto the target architecture, and guarantees that the logical semantics is preserved (w.r.t. both functionality and timing). However, a GIOTTO program might be *over constrained* when it does not permit any execution consistent with the platform constraints. In such a case, the compiler rejects it as non valid. In contrast, our method always produces an executable distributed program. Then, determining whether this program meets the desired timing constraints is left to the user. Since we use finite deterministic automata, computing such a worst case execution time is feasible, although still subject to the intrinsic non-determinism of modern processors (multi-level cache, pipeline, instruction level parallelism, and so on).

Another related work is Real-Time Workshop's (RTW) distributed code generation for SIMULINK programs [The MathWorks, Inc. 1999]. In SIMULINK, one can

assign different rates to different blocks, in order to deal with long duration blocks. This approach is similar to clocks in LUSTRE, except that these rates are *static* whereas clocks are *dynamic*. Figure 23(a) is an example of such a program: the WCET of task B is five times longer than the WCET of task A, but accordingly, the rate of task A is five time faster than the rate of task B. RTW produces a schedule where task B is preempted by the successive instances of task A, which is faster and has therefore a higher priority: this schedule is shown in Figure 23(b). Due to the successive preemptions of task B by task A, task B is sliced into several chunks: B1, B2, B3, B4, and B5. The solid arrows from the successive instances of A to B1, B2, and so on, depict communications. The dashed arrow going down from B1 to A depict the beginning of the preemption of B by A. The dashed arrow going up from A to B1 depict the end of the preemption.

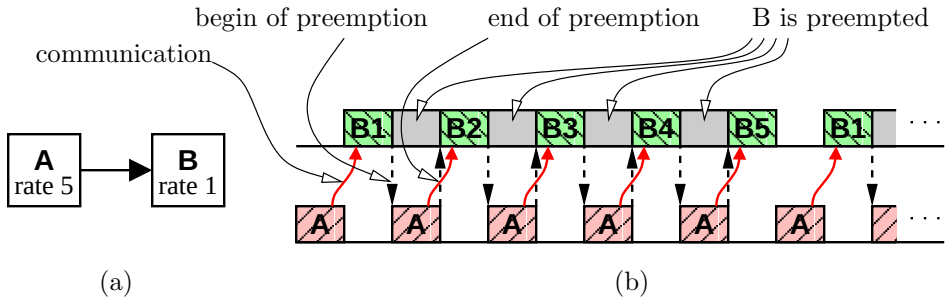

Fig. 23. (a) Two SIMULINK blocks; (b) The schedule produced by RTW.

The communication from A to B raises a problem: how can the data produced by A during its first cycle, and not the subsequent data, be used by B? As we can see in Figure 23(b), task A sends its output *at each of its cycles*, while task B only expects an input data *at the beginning of its own cycle*. The solution involves adding a *zero oracle* between A and B, which inherits the priority of A and the rate of B, as shown in Figure 24(a). In the new schedule of Figure 24(b), only one data is sent and received. When the communication takes place from a low rate block towards a high rate one, the solution involves adding a *unit delay* between the two blocks.

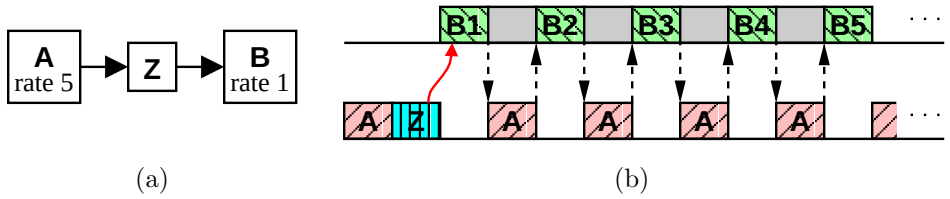

Fig. 24. (a) Two SIMULINK blocks with a zero oracle; (b) The new schedule.

Another related work is ESTEREL's *asynchronous tasks* mechanism [Paris 1992]. Basically, a synchronous ESTEREL program can launch an asynchronous task by means of a dedicated output signal (called `exec`), and then be warned of its termination by means of a dedicated input signal (called `return`). Although not as powerful and flexible as rates, this mechanism allows long duration tasks to be taken into account. However, they are handled *externally* w.r.t. the synchronous program, while our method allows them to be handled *inside* the OC program.

Also, there has been a lot of work done on the automatic distribution of SIGNAL/POLYCHRONY programs [Maffeïs 1993; Aubry et al. 1996; Benveniste et al. 1998; Benveniste et al. 2000]. Distribution in SIGNAL is performed by first compiling the program into a *hierarchical data-flow graph* with *conditional dependency equations*, called a Synchronized Data-Flow Graph (SDFG). Vertices of this SDFG are signals and variables, while the edges are labeled with clocks. Therefore, each vertex is located in a hierarchy of clocks. In the most general case, the set of all the program's clocks form a *forest* instead of a *tree*. This means that the program *does not* have a base clock, which raises code generation problems since the periodic rate of the program cannot be statically determined. Yet, for a wide class of programs, the forest is reduced to a single tree and the program does have a base clock. Such programs are called *endochronous*. Once the SDFG is built, communications are inserted in it, then subgraphs are extracted corresponding to different computing locations. It is then checked that two programs obtained in this manner are *isochronous*, meaning that their synchronous composition is equivalent to their asynchronous composition. Finally, sequential code is generated from each subgraph. However, this method does not directly perform the rate desynchronization (or the clock-driven distribution) of SIGNAL program, meaning that the user is required to partition *manually* the source program into clock-wise fragments [Le Guernic 2003]. An interesting perspective would be to apply our rate desynchronization method to SDFGs.

Finally, a recent article [Caspi et al. 2003] describes a distributed implementation of LUSTRE programs over Time-Triggered Architectures (TTA) [Kopetz 1997]. TTA includes a synchronous bus connected to all the processors, which sends them a global fault-tolerant clock. The authors introduce extensions of LUSTRE to annotate a program in order to define a periodic clock (`periodic_cl`), to specify where a block should be executed (`location`), what its basic clock should be (`basic_period`), and to specify its WCET (`exec_time`). Then, the authors describe their implementation of LUSTRE over TTA, using a scheduler that, starting from the data dependencies partial order given by the LUSTRE compiler, has to solve a multiperiod multiprocessor scheduling problem. If no solution is found, the graph is refined. However, for long duration tasks to be scheduled, it is necessary to have a partition of the program into smaller blocks (to be expressed in the source program by the user himself), while our method computes this partition automatically.

5.6 Concluding remarks

To conclude, we would like to stress the original contributions of our work. We provide a method and tool to automatically distribute programs having a very general control structure (a finite state automaton with a DAG of actions in each state) according to the rates of the inputs and outputs, hence avoiding the manual partition of an entire system into several tasks with different periods. The user only

needs to partition the set of the program's rates. This *rate desynchronization* allows long duration tasks to be taken into account inside such programs. That is, we are able to produce automatically distributed programs where the parts that perform slow computations do not impair the global reaction time of the program. This feature is crucial for embedded reactive programs having tight temporal constraints. Once the distribution is done, the Liu & Layland conditions can be checked to choose the most suited priority policy.

We have also shown how our rate desynchronization method can be applied to distribute a LUSTRE program according to its clocks, as well as to distribute an ESTEREL program where the computations are sufficiently decoupled, into parts that can be run at different rates, exactly as if the ESTEREL program had clocks.

Acknowledgments

Many thanks to Prof. Edward Lee (UC Berkeley) for a very helpful explanation of Real-Time Workshop's distributed code generation, to Albert Benveniste and Paul Le Guernic (INRIA/IRISA) for interesting discussions on the SIGNAL approach to automatic distribution, and to the anonymous reviewers for their comments and suggestions.

REFERENCES

AUBRY, P., LE GUERNIC, P., AND MACHARD, S. 1996. Synchronous distributions of Signal programs. In *Hawaii International Conference on System Sciences, HICSS'96*. IEEE, Honolulu, USA, 656–665.

BENVENISTE, A. ET AL. 1998. Safety critical embedded systems design: the SACRES approach. Tutorial at the Symposium on Formal Techniques in Real-Time and Fault Tolerant systems, FTRTFT'98.

BENVENISTE, A., CAILLAUD, B., AND LE GUERNIC, P. 2000. Compositionality in dataflow synchronous languages: Specification and distributed code generation. *Information and Computation 163*, 125–171.

BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. 2003. The synchronous languages twelve years later. *Proceedings of the IEEE 91,* 1 (Jan.), 64–83. Special issue on embedded systems.

BERGERAND, J.-L. AND PILAUD, E. 1988. SAGA: a software development environment for dependability in automatic control. In *International Conference on Computer Safety, Reliabilitiy, and Security, SAFECOMP'88*. Pergamon Press, Fulda, Germany.

BERRY, G. AND GONTHIER, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

CAILLAUD, B., CASPI, P., GIRAULT, A., AND JARD, C. 1997. Distributing automata for asynchronous networks of processors. *European Journal of Automation (RAIRO-APII-JESA) 31,* 3, 503–524. Research report Inria 2341.

CASPI, P. 1992. Clocks in data-flow languages. *Theoretical Computer Science 94*, 125–140.

CASPI, P., CURIC, A., MAIGNAN, A., SOFRONIS, C., TRIPAKIS, S., AND NIEBERT, P. 2003. From Simulink to Scade/Lustre to TTA: A layered approach for distributed embedded applications. In *International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES'03*. ACM, San Diego, USA, 153–162.

CASPI, P., FERNANDEZ, J.-C., AND GIRAULT, A. 1995. An algorithm for reducing binary branchings. In *15th Conference on the Foundations of Software Technology and Theoretical Computer Science, FST&TCS'95*, P. Thiagarajan, Ed. LNCS, vol. 1026. Springer-Verlag, Bangalore, India, 279–293.

Caspi, P. and Girault, A. 1995. Execution of distributed reactive systems. In *1st International Conference on Parallel Processing, EURO-PAR'95*, S. Haridi, K. Ali, and P. Magnusson, Eds. LNCS, vol. 966. Springer-Verlag, Stockholm, Sweden, 15–26.

Caspi, P., Girault, A., and Pilaud, D. 1999. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Trans. on Software Engineering 25,* 3 (May), 416–427.

Caspi, P. and Pouzet, M. 1996. Synchronous Kahn networks. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'96*. ACM Press, Philadelphia, USA.

Colaço, J.-L. and Pouzet, M. 2003. Clocks as first class abstract types. In *International Conference on Embedded Software, EMSOFT'03*, R. Alur and I. Lee, Eds. LNCS, vol. 2855. Springer-Verlag, Philadelphia, USA, 134–155.

Fernandez, J.-C. and Mounier, L. 1991. "On the fly" verification of behavioural equivalences and preorders. In *Workshop on Computer-Aided Verification, CAV'91*, K. Larsen, Ed. LNCS. Springer-Verlag, Aalbord, Denmark.

Gupta, R., Pande, S., Psarris, K., and Sarkar, V. 1999. Compilation techniques for parallel systems. *Parallel Computing 25,* 13, 1741–1783.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. 1991. The synchronous data-flow programming language Lustre. *Proceedings of the IEEE 79,* 9 (Sept.), 1305–1320.

Harel, D. and Pnueli, A. 1985. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag.

Henzinger, T., Horowitz, B., and Kirsch, C. 2003. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE 91*, 84–99.

Kopetz, H. 1997. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers.

Le Guernic, P. 2003. Personnal communication.

Le Guernic, P., Gautier, T., Borgne, M. L., and Lemaire, C. 1991. Programming real-time applications with Signal. *Proceedings of the IEEE 79,* 9 (Sept.), 1321–1336.

Liu, C. and Layland, J. 1973. Scheduling algorithms for multiprogramming in hard real-time environnement. *Journal of the ACM 20,* 1 (Jan.), 46–61.

Maffeïs, O. 1993. Ordonnancements de graphes de flots synchrones ; application à la mise en œuvre de Signal. Ph.D. thesis, University of Rennes I, Rennes, France.

Milner, R. 1980. *A Calculus of Communicating Systems*. LNCS, vol. 92. Springer-Verlag.

Paris, J.-P. 1992. Exécution de tâches asynchrones depuis Esterel. Ph.D. thesis, University of Nice, Nice, France.

Plaice, J. and Saint, J.-B. 1987. *The Lustre-Esterel Portable Format*. Inria, Sophia-Antipolis, France. User Manual.

Salpétrier, F. 2002. Interface graphique utilisateur pour la répartition de programmes Lustre dirigée par les horloges. M.S. thesis, ESISAR, Valence, France.

Sha, L., Rajkumar, R., and Lehoczky, J. 1990. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. on Computers 39*, 1175–1185.

The MathWorks, Inc. 1999. *Real-Time Workshop User's Guide, Version 3*.