Chapter 2: Automatic distribution of Lustre and Esterel synchronous programs

Alain Girault
(Joint work with Paul Caspi)
(and with Clément Ménier for circuits)

January 2014 – ENS Lyon

Outline

Outline

Context

Emebdded and reactive systems are distributed :

- physical location of sensors and actuators
- fault-tolerance
- performance improvement

In general, distribution is driven by the user

Synchronous programming languages are concurrent

Context

Emebdded and reactive systems are distributed :

- physical location of sensors and actuators
- fault-tolerance
- performance improvement

In general, distribution is driven by the user

Synchronous programming languages are concurrent

But :

expression parallelism $\neq$ execution parallelism

Separate programming of each computing location

Advantages

- efficiency of the code
- divide and conquer approach

Inconvenients

- no global view of the system
- no semantics of communication
- debugging a distributed program is difficult

Asynchronous parallel programming languages

E.g. : Ada, Occam, multi-threaded Java, ...

Advantages

- global view
- debugging on the source code

Inconvenients

- the interleaving semantics is non-deterministic
- debugging must be performed on non-deterministic sequential object code

[E.A. Lee, The problem with threads]

Synchronous parallel programming languages

E.g. : Esterel, Lustre, Signal/Polychrony, Heptagon, Prelude, ...

Advantages

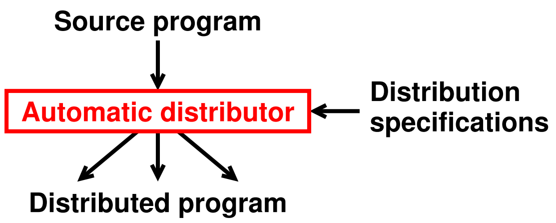
- global view
- debugging on the source code
- debugging on deterministic sequential object code

Inconvenients

- how to generate distributed code ?

Automatic distribution

To benefit from the advantages of synchronous programming, one must generate automatically the corresponding distributed code.



Source program

Automatic distributor

Distribution specifications

Distributed program

Distribution specifications : N computing locations $\Longrightarrow$ particion of the set of inputs / outputs into N subsets.

➪ Driven by the physical location of the sensors and actuators
(We do not seek the best performances nor the maximal parallelism)

Outline

Direct source code distribution (1)

Algorithm

- Cut the source program into N fragments
- Compile separately each fragment
- Make the N fragments communicate harmoniously

Direct source code distribution (1)

Algorithm

- Cut the source program into N fragments
- Compile separately each fragment
- Make the N fragments communicate harmoniously

This is the ideal solution

Direct source code distribution (1)

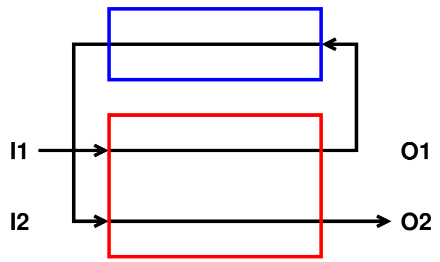
Algorithm

- Cut the source program into *N* fragments
- Compile separately each fragment
- Make the *N* fragments communicate harmoniously

This is the ideal solution

But in general it does not work

A counter-example (1)



```
node MAIN (I1:int) returns (O2:int);
var O1,I2:int;
let
O1 = I1;
O2 = I2;
I2 = O1;
tel;
```

```
I2 = O1;
```

```
O1 = I1;
O2 = I2;
```

A counter-example (2)

Compiling a concurrent program (e.g., Lustre) into sequential code means sequentializing it !

The red fragment can be sequentialized in two ways :

main program	blue fragment	red fragment
`O2:=I1;`		`O1:=I1;`
	`I2:=O1;`	
		`O2:=I2;`

A counter-example (2)

Compiling a concurrent program (e.g., Lustre) into sequential code means sequentializing it !

The red fragment can be sequentialized in two ways :

main program	blue fragment	red fragment
`O2:=I1;`	`O1:=rcv(R);`	`O1:=I1;`
	`I2:=O1;`	`snd(B,O1);`
	`snd(R,I2);`	`I2:=rcv(B);`
		`O2:=I2;`

A counter-example (2)

Compiling a concurrent program (e.g., Lustre) into sequential code means sequentializing it !

The red fragment can be sequentialized in two ways :

main program	blue fragment	red fragment
O2:=I1;	O1:=rcv(R); I2:=O1; snd(R,I2);	O1:=I1; snd(B,O1); I2:=rcv(B); O2:=I2;
O2:=I1;	I2:=O1;	O2:=I2; O1:=I1;

A counter-example (2)

Compiling a concurrent program (e.g., Lustre) into sequential code means sequentializing it !

The red fragment can be sequentialized in two ways :

main program	blue fragment	red fragment
O2:=I1;	O1:=rcv(R); I2:=O1; snd(R,I2);	O1:=I1; snd(B,O1); I2:=rcv(B); O2:=I2;
O2:=I1;	O1:=rcv(R); I2:=O1; snd(R,I2);	I2:=rcv(B); O2:=I2; O1:=I1; snd(B,O1);

Direct source code distribution (2)

Algorithm

- Cut the source program into N fragments
- For each fragment 1 to N :
  - compile fragment i, taking into account the scheduling constraints C_1 to C_{i-1}
  - synthesize the scheduling constraints C_i for the next fragments

---

Direct source code distribution (2)

Algorithm

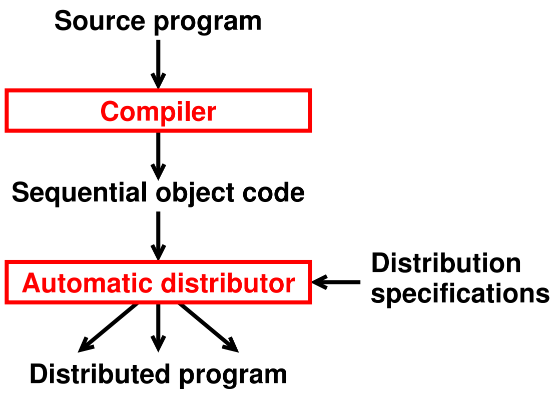
- Cut the source program into N fragments
- For each fragment 1 to N :
  - compile fragment i, taking into account the scheduling constraints C_1 to C_{i-1}
  - synthesize the scheduling constraints C_i for the next fragments

Solution adopted in Signal

The code distribution algorithm must perform the causality analysis at the same time as the distribution

Problem : in which orderr must the fragments be compiled ?

Object code distribution



Source program

Compiler

Sequential object code

Automatic distributor

Distribution specifications

Distributed program

- ➪ The source program is debugged first
- ➪ The causality analysis is performed by the compiler
- ➪ The method can be common to several synchronous languages

---

OC automaton

- Common format to the Lustre and Esterel compilers
- Finite state automaton with a DAG of actions in each state
- One reaction of the program = one transition of the automaton
- Purely sequential control flow
- Explicit and static control structure

SC circuit

- Output format of the Esterel compiler
- Sequential circuit with a finite memory to drive a table of actions on data types
- One reaction of the program = one clock cycle of the circuit
- Parallel control flow
- Implicit and dynamic control structure
- Opens up possibilities to do hardware/software codesign

Outline

1. Context and overview
2. The different distribution approaches
3. OC code distribution
4. SC code distribution
5. CP code distribution

Structure of the OC code

An OC program handles signals and variables :

- signal = input/output of the source program
- variable = associated to valued signals and local variables

The nodes of the DAG can be :

- Root : Implicit read of the input signals
- Unary node :
  - Variable assignment : `x:=exp`
  - Output signal emission : `output y`
  - External procedure call : `call p`
- Binary node : binary test : `if`, `present`
- Leaf : change state : `goto s`

A running example in OC

```
input ck,x:integer;
output y,z:integer;

State 0

go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

Distribution directives

The user wants N computing sites

➪ Partition of the set of inputs/outputs of the program into N subsets $V_i \quad (i = 1..N)$

Running example :

- Site 0 : $V_0 = \{ck, x, z\}$
- Site 1 : $V_1 = \{y\}$

OC distribution algorithm

1. Duplicate the sequential code on each computing location
2. Assign a location to each variable and action
3. On each computing location, do :
   1. Prune useless actions
   2. Insert communications
   3. Insert synchronizations

Classical notations :

- use(A) = {variables used by node A}
- def(A) = {variables modified by node A}

The OC running example

State 0

```
go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```
go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```
go(ck,x);
if (ck) then
  y:=calcul(x);
  output(y);
else
  z:=x;
  output(z);
endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
if (ck) then


else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```

if (ck) then
  y:=calcul(x);
  output(y);
else


endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
if (ck) then


else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```

if (ck) then
  y:=calcul(x);
  output(y);
else


endif
goto 0;
```

Inter-cite data dependencies !

➪ Need to insert communications.

Choice of the communication primitives

Rendezvous (Ada, OCCAM)

- asynchronous but synchronizing
- incur unnecessary delays

FIFOs

- truly asynchronous
- send and receive delayed
- send and receives must be performed in the same order

Communication primitives

Send

- `snd(j,x)` insert value `x` in the FIFO connected to site `j`
- non-blocking
- (could be blocking when FIFO is full for synchronization)

Receive

- `y:=rcv(i)` extracts the head value from the FIFO connected to site `i` and assigns it to variable `y`
- blocking when the FIFO is empty

Communication insertion algorithm

Sends

Compute at each node of the DAG the sets E^s_{need} of variables needed by `s` :

1. Traverse the DAG backward starting from the leaves
2. For each `x` $\in$ `use(A)`, if `x` $\notin V_s$ then $E^s_{need} := E^s_{need} \cup \{x\}$
3. For each `y` $\in$ `def(A)`, if `y` $\in E^s_{need}$ then insert a `snd(s,x)` in the DAG of site `t`

Receives

Compute at each node of the DAG the ordered sets $Q^{(s,t)}_{fifo}$ of variables sent by `s` to `t` :

1. Traverse the DAG forward starting from the root
2. For each `snd(t,x)` insert `x` in $Q^{(s,t)}_{fifo}$
3. For each `x` $\in$ `use(A)`, if `x` $\notin V_s$ then insert a `x:=rcv(s)` in the DAG of site `t`

The OC running example

State 0 – Site 0

```
go(ck,x);

if (ck) then



else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```


if (ck) then

  y:=calcul(x);
  output(y);
else



endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
snd(1,ck);
if (ck) then
  snd(1,x);


else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```


if (ck) then

  y:=calcul(x);
  output(y);
else



endif
goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);
snd(1,ck);
if (ck) then
  snd(1,x);


else
  z:=x;
  output(z);
endif
goto 0;
```

State 0 – Site 1

```

ck:=rcv(0);
if (ck) then
  x:=rcv(0);
  y:=calcul(x);
  output(y);
else



endif
goto 0;
```

Resynchronization

One computing location could be purely a producer of values for another location (e.g., site 0)

➪ Can lead to unbounded FIFOs

The initial centralized program follows a notion of cycle / reaction (= one transition of the OC automaton)

➪ What is the meaning in the distributed case ?

Resynchronization

One computing location could be purely a producer of values for another location (e.g., site 0)

➪ Can lead to unbounded FIFOs

The initial centralized program follows a notion of cycle / reaction (= one transition of the OC automaton)

➪ What is the meaning in the distributed case ?

Resynchronization methods :

- Strong resynchronization
- Weak resynchronization

Strong synchronization

No delay at all between any two computing location :

➪ All computing location must execute synchronously the same automaton reaction

➪ A synchronization must occur at the end of each reation

- A token circulating twice between all N nodes : $2 \times N$ synchronization messages
- A rendezvous between all N nodes : $N \times (N-1)$ synchronization messages

---

Weak synchronization

At most one time lag between any pair of computing locations

Weak "total" sychronization :

- At least one message exchange between any two locations at each reaction
- Built upon the existing sends and receives

Weak synchronization

At most one time lag between any pair of computing locations

Weak "total" sychronization :

- At least one message exchange between any two locations at each reaction
- Built upon the existing sends and receives

Weak "if needed" synchronization

- Only between locations that already communicate with each other
- Only during the reactions where they do communicate

Weak synchronization

At most one time lag between any pair of computing locations

Weak "total" sychronization :

- At least one message exchange between any two locations at each reaction
- Built upon the existing sends and receives

Weak "if needed" synchronization

- Blocking `snd` to implement bounded capacity FIFOs
- Relaxed form of resynchronization where there can be N time lags

---

The OC running example

State 0 – Site 0

```
go(ck,x);

snd(1,ck);
if (ck) then
  snd(1,x);



else
  z:=x;
  output(z);
endif

goto 0;
```

State 0 – Site 1

```



ck:=rcv(0);
if (ck) then
  x:=rcv(0);
  y:=calcul(x);
  output(y);
else


endif

goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);

snd(1,ck);
if (ck) then
  snd(1,x);



else
  z:=x;
  output(z);
endif

goto 0;
```

State 0 – Site 1

```


snd_void(0);
ck:=rcv(0);
if (ck) then
  x:=rcv(0);
  y:=calcul(x);
  output(y);
else


endif

goto 0;
```

The OC running example

State 0 – Site 0

```
go(ck,x);

snd(1,ck);
if (ck) then
  snd(1,x);



else
  z:=x;
  output(z);
endif
rcv_void(1);
goto 0;
```

State 0 – Site 1

```


snd_void(0);
ck:=rcv(0);
if (ck) then
  x:=rcv(0);
  y:=calcul(x);
  output(y);
else


endif

goto 0;
```

Discussion

Benefits :

- It works
- There is a formal correctness proof [Caillaud, Caspi, et al, 1994], based on semi-commutations and transition systems labelled with partial orders

Discussion

Benefits :

- It works
- There is a formal correctness proof [Caillaud, Caspi, et al, 1994], based on semi-commutations and transition systems labelled with partial orders

Drawbacks :

- The OC automaton must be generated first, which suffers from the well known state space explosion
- The distribution is strict : all the computing locations must have the same rate
- The communications must be lossless

Outline

1. Context and overview
2. The different distribution approaches
3. OC code distribution
4. **SC code distribution**
5. CP code distribution

Automatic Production of Globally Asynchronous Locally Synchronous Systems

Alain GIRAULT INRIA Rhône-Alpes

and

Clément MÉNIER ENS Lyon

Outline

1. GALS Systems
2. Related Work
3. Program model
4. Our method in details
5. Perspectives

GALS Systems

Acronym for "Globally Asynchronous Locally Synchronous"

In software : paradigm for composing blocks and making them communicate asynchronously
➪ Used in embedded systems

In hardware : circuits designed as sets of synchronous blocks communicating asynchronously
➪ No need to distribute the clock ⟹ saves power

Our goal : automatically obtain GALS systems from a centralised program

---

Automatically Distributed Programs

Why distribute ?

➪ physical constraints, fault-tolerance, performance...

Advantages of automatic distribution :

- less error-prone than by hand
- possibility to debug & validate before distribution
- ...

Related Work

The closest is *Berry & Sentovich'2000* :
"Implementation of constructive synchronous circuits as a network of CFSMs in POLIS"

Main differences with our work :

1. Partitioning of the circuit into N clusters is by hand
   Our partitioning is automatic
2. They partition the circuit, that is the control part
   We partition the data part and replicate the control part

Program Model : Synchronous Circuits (1)

Program = synchronous sequential circuit driving a table of actions

A control part and a data part :

- Control part = synchronous sequential boolean circuit
- Data part = table of external actions
  - ➪ manipulate inputs, outputs, and typed variables (integers, reals...)

A program has a set of input and output signals

Signals can be pure or valued

Valued signals are associated to a local typed variable

It can be obtained from Esterel ⟹ so called SC internal format

VHDL code can be generated from it

Program Model : Synchronous Circuits (2)

The basic elements of SC circuits :

- standard : computes a Boolean expression (regular gates of the circuit)
- action : triggers an action from the table (the control is passed)
- ift : triggers a test from the table and assigns to the wire the result of the test
- input : takes the value of the presence Boolean of the signal and updates the value of the associated variable (if the input is valued)
- output : triggers an output action from the table
- register : a latch with an initial value

The circuit registers

The encode the internal state of the circuit :

- One boot register
- One loop register
- Several regular registers

A valuation of the register vector corresponds to one state of the OC automaton

A reaction of the program is one clock cycle

There are several contorl paths

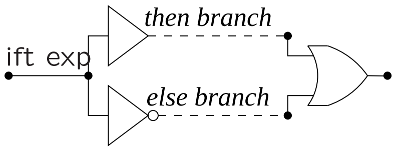
Program Model : Properties

The control structure is :

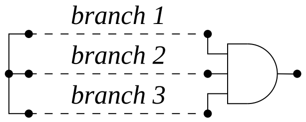
- Parallel : there are several control paths
- Implicit : the state is coded in the registers
- Dynamic : the control depends on the data

Important property : any given variable can only be modified in one parallel branch (same as in Esterel)

Control structures



Exactly like a binary branching (so dealt with as before)

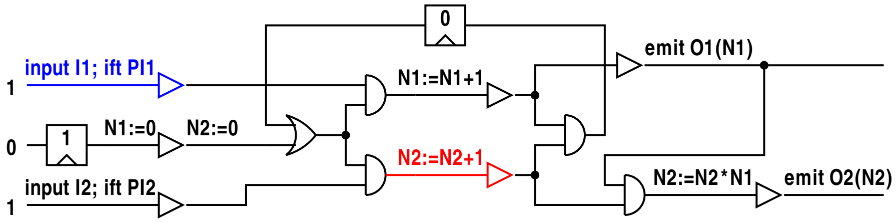


It is not possible to tell in which order the actions are performed

Impossible to simulate at compile-time the state of the FIFO queues to insert the receive operations

One FIFO per variable

An Example : `FOO`



`input I1; ift PI1`
`input I2; ift PI2`
`N1 := 0`

`N2 := N2 + 1`
`N2 := N2 * N1`
`emit O1(N1)`

`N2 := 0`
`emit O2(N2)`
`N1 := N1 + 1`

Distribution Method

1. Design a centralised system

2. Compile it into a single synchronous circuit

3. Distribute it into N communicating synchronous circuits

We focus here on the point 3 : the automatic distribution

Distribution Specifications

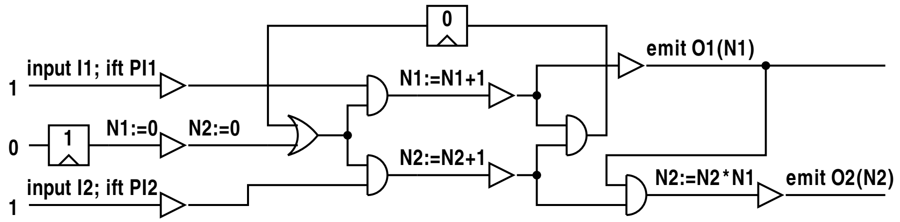
Must be provided by the user :

- The desired number of computing locations

- The localisation of each input and output

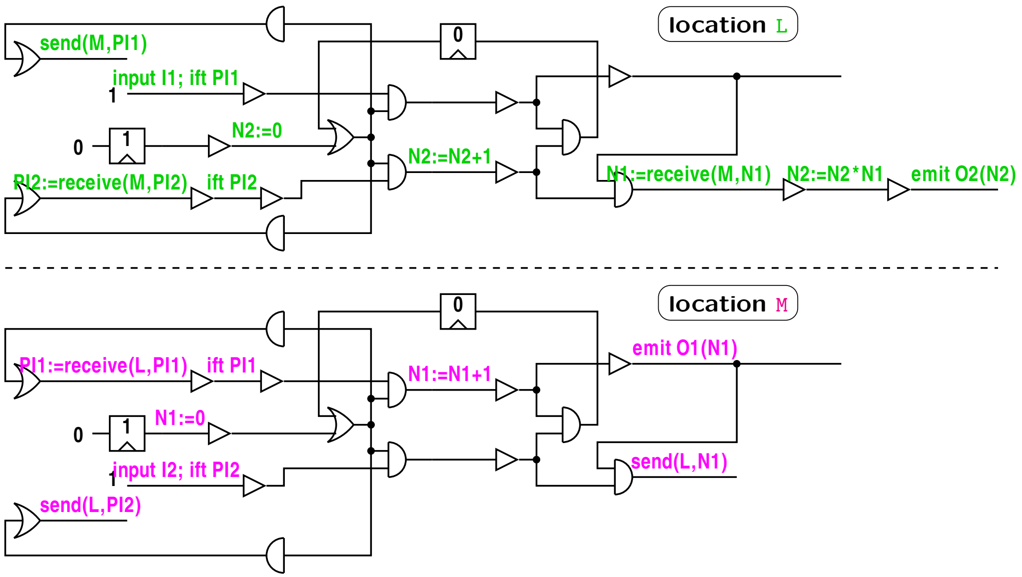
  ➪ derived from the physical localisation of the sensors and actuators

Distribution Specifications of `FOO`



location L	location M
I1,O2	I2,O1

Where are we Heading ?



Distribution Algorithm : Principle

Based on past work : *Caspi, Girault, & Pilaud'1999*

➪ Replicate the control part and partition the data part

1. Localise each action to get N virtual circuits
2. Solve the distant variables problem for each virtual circuit
3. Project each virtual circuit to get one actual circuit
4. Solve the distant inputs problem

We obtain N circuits communicating harmoniously
➪ without inter-blocking and with the same functional behaviour

---

Communication Primitives

Asynchronous communications

➪ Two FIFO queues associated with each pair of locations and each variable

➪ Each queue is identified by a triplet $\langle \mathtt{src}, \mathtt{var}, \mathtt{dst} \rangle$

Two communication primitives :

- On location `src` : `send(dst,var)` non blocking
- On location `dst` : `var:=receive(src,var)` blocking when empty

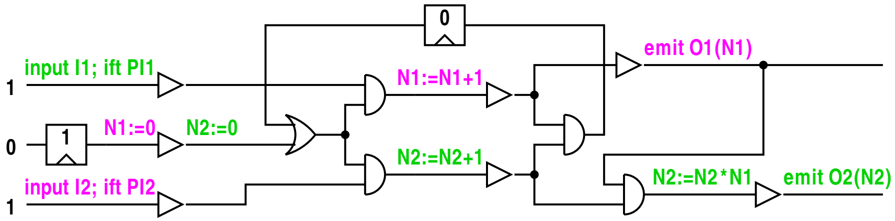
Localisation of the Actions

Only the data part is partitioned : the control part is replicated

loc.	action
L	input I1; ift PI1
M	input I2; ift PI2
M	N1 := 0

loc.	action
L	N2 := N2 + 1
L	N2 := N2 ∗ N1
M	emit O1(N1)

loc.	action
L	N2 := 0
L	emit O2(N2)
M	N1 := N1 + 1



Two Problems to Solve

1. Distant variables problem :

➪ Not computed locally

➪ We add send and receive actions

2. Distant inputs problem :

➪ Not received locally

But : input signals convey two informations : value and presence

And : an ift net is required to modify the control flow according to the input's presence

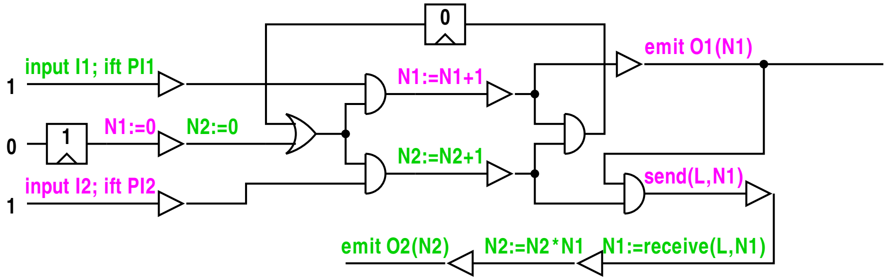
➪ We add input simulation blocks

Solving the Distant Variables Problem

We apply a simple algorithm to solve the data dependencies to each buffered path (sequential path) :

1. Isolate a buffered path and mark its root and tail
2. Insert the send actions in the buffered path
   - ➪ Traverse the path backward to insert the send actions asap
3. Insert the receive actions in the buffered path
   - ➪ Traverse the path forward to insert the receive actions alap
4. Proceed to the unmarked successor nets of the tail

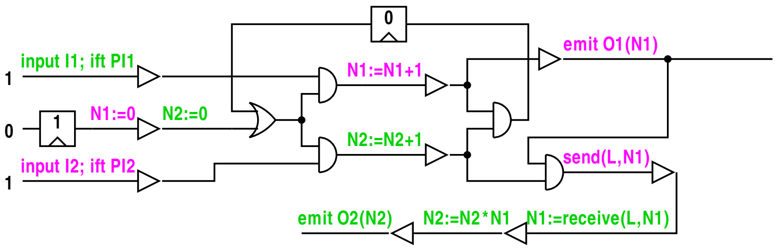
Partial Result for FOO



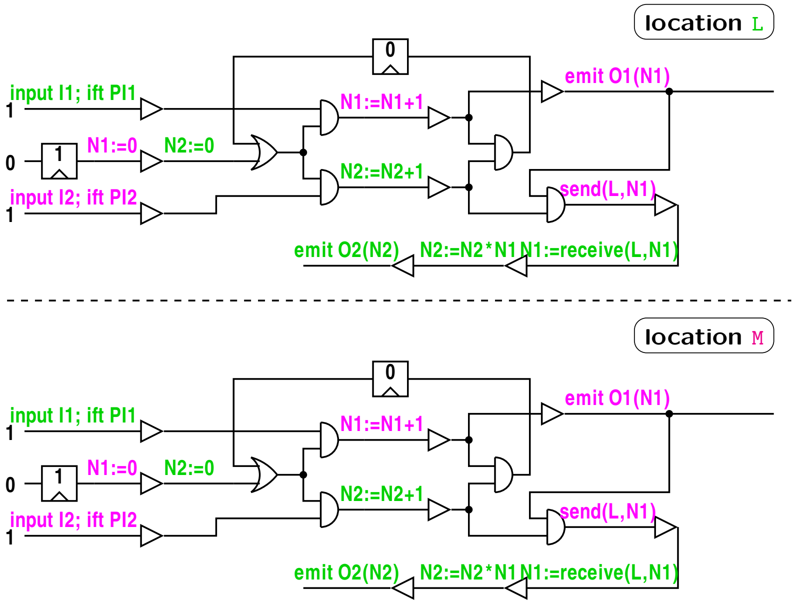
This is still one circuit representing two virtual circuits

The next step is to project onto two actual circuits

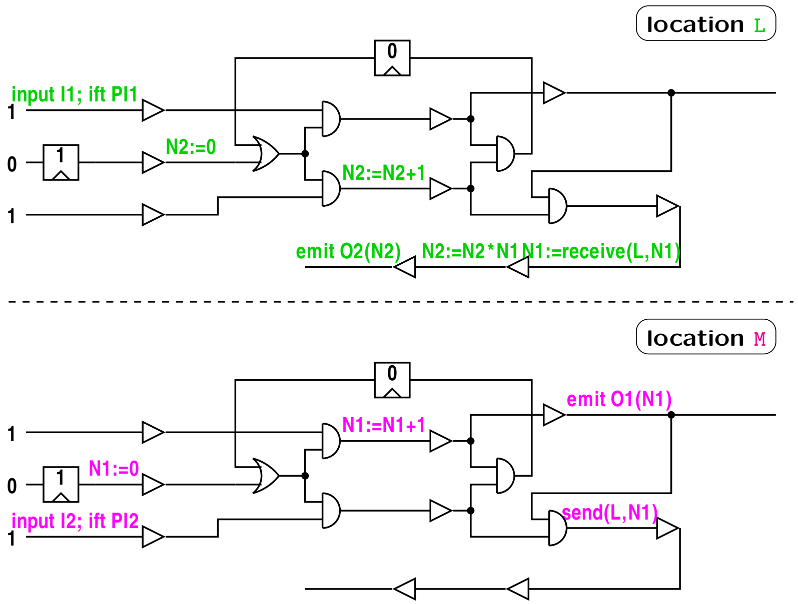
Projection for FOO



Projection for FOO



Projection for FOO



Solving the Distant Inputs Problem

Reminder : input signals convey two informations : the value and the presence

And : an `ift` net is required to modify the control flow according to the input's presence

Our goal is to send the presence information only to those computing locations that need them :

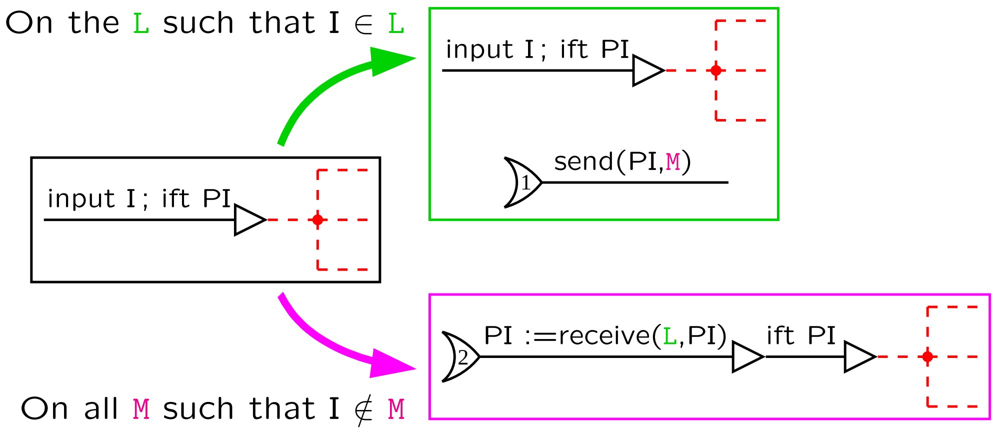
1. Detect the impure input-dependent nets and their needed inputs

   ➪ Circuit traversal to compute for each net the set
   `SInput` $=$ {`needed inputs`}

2. Create the simulation blocks for the input nets

3. Connect the nets detected at step 1 to the required simulation blocks

Creation of the Input Simulation Blocks



On the $\mathtt{L}$ such that $\mathrm{I} \in \mathtt{L}$

input I ; ift PI

input I ; ift PI

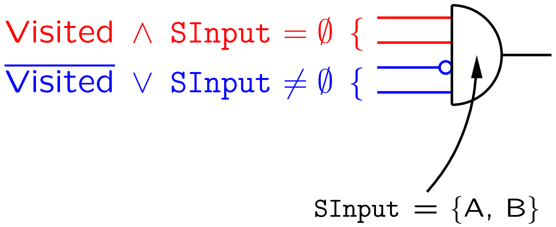
send(PI,M)

PI :=receive(L,PI)

ift PI

On all $\mathtt{M}$ such that $\mathrm{I} \notin \mathtt{M}$

Connection of the Input Simulation Blocks

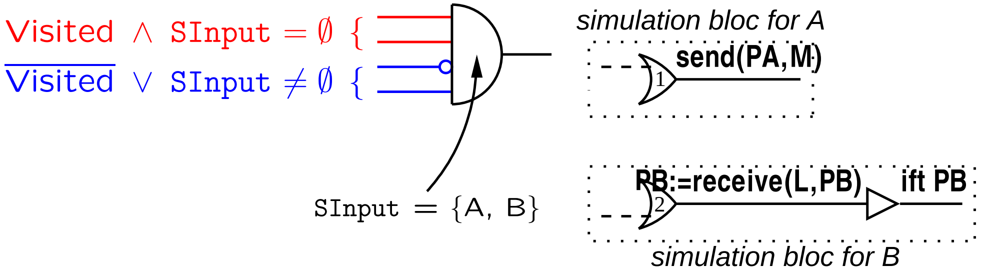


$\text{Visited} \wedge \mathtt{SInput} = \emptyset\ \{$

$\overline{\text{Visited}} \vee \mathtt{SInput} \neq \emptyset\ \{$

$\mathtt{SInput} = \{\text{A, B}\}$

Connection of the Input Simulation Blocks



$\text{Visited} \wedge \mathtt{SInput} = \emptyset\ \{$

$\overline{\text{Visited}} \vee \mathtt{SInput} \neq \emptyset\ \{$

$\mathtt{SInput} = \{\text{A, B}\}$

simulation bloc for A

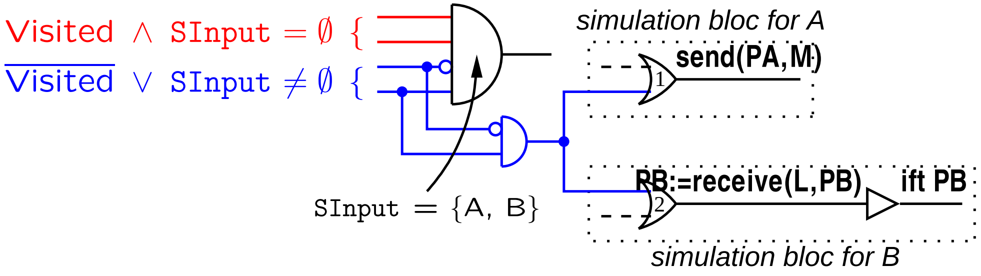
send(PA,M)

PB:=receive(L,PB)

ift PB

simulation bloc for B

Connection of the Input Simulation Blocks



$\text{Visited} \wedge \mathtt{SInput} = \emptyset\ \{$

$\overline{\text{Visited}} \vee \mathtt{SInput} \neq \emptyset\ \{$

$\mathtt{SInput} = \{\text{A, B}\}$

simulation bloc for A

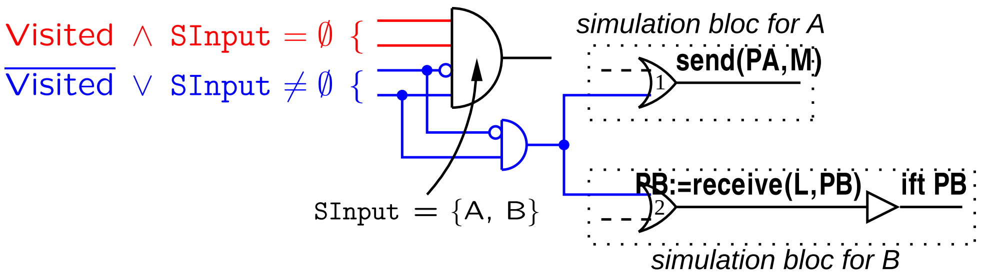
send(PA,M)

PB:=receive(L,PB)

ift PB

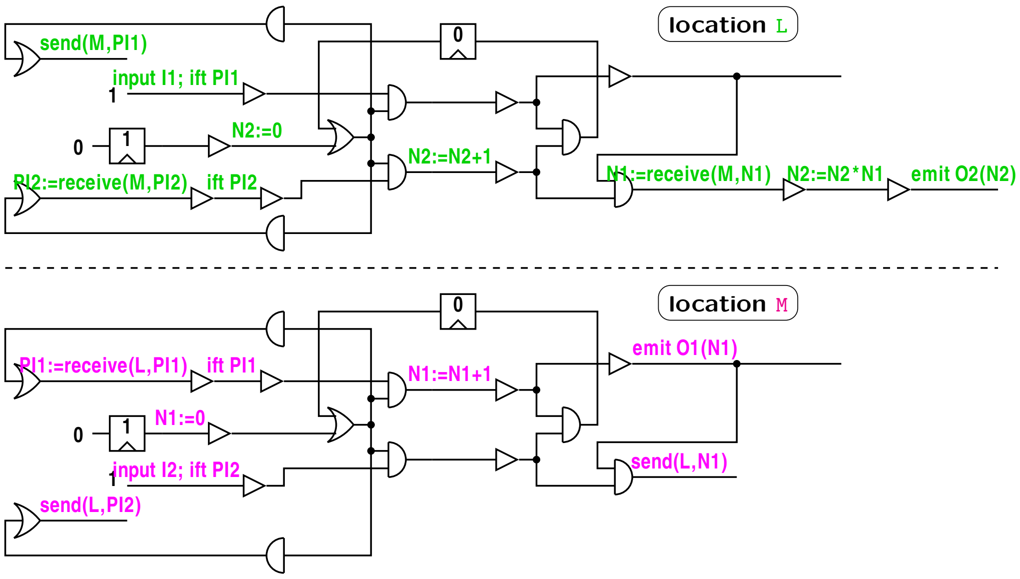
simulation bloc for B

Connection of the Input Simulation Blocks



Connection of an OR gate is similar

Final Result for FOO



Conclusion

This methods is interesting only if the data part is big (because the control part is replicated)

Open directions : hardware/software codesign, post-distribution optimisations, …

The most interesting perspective is to mix this approach with *Berry & Sentovich'2000* :

- Accepting as inputs cyclic constructive circuits
- Automatic partitioning of the circuit into N clusters
- Partitioning both the data part and the control part

Outline

1. Context and overview
2. The different distribution approaches
3. OC code distribution
4. SC code distribution
5. CP code distribution

Modern compiling methods for Esterel

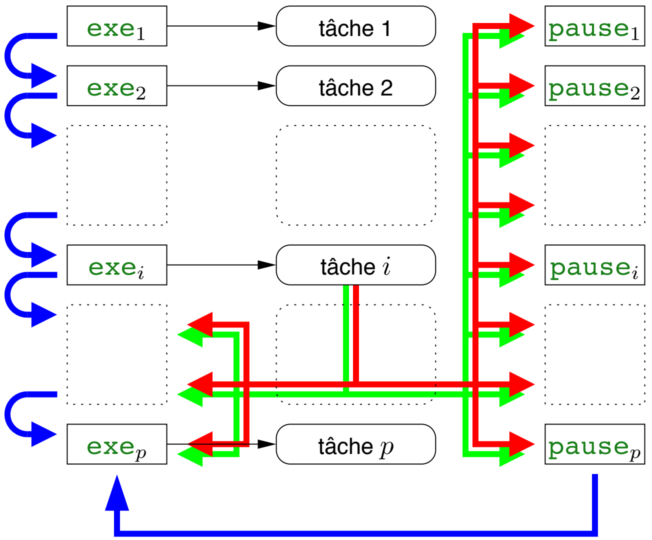
- [Weil, Bertin, Closse, Poize, Venier & Pulou, CASES'00]
- [Edwards, CODES'99]
- [Potop, PhD'02] and [Potop, Edwards & Berry, 2007]

Common principle

- Linked list of control points
- Each control point is attached to a block of sequential code
- At each reaction, the list is traversed to execute only the active control points
- A sequential block can activate another block, but only further in the list or for the next reaction

---

CPREP within SAXO-RT for ESTEREL



Each `tâche` i is a DAG of actions

Distribution algorithm of CPREP

1. Replicate the control structure (`exe` and `pause` vectors) onto each computing location

---

Distribution algorithm of CPREP

1. Replicate the control structure (`exe` and `pause` vectors) onto each computing location
2. Apply the OCREP algorithm to the DAG of each `tâche` i

Distribution algorithm of CPREP

1. Replicate the control structure (`exe` and `pause` vectors) onto each computing location
2. Apply the OCREP algorithm to the DAG of each `tâche` i

Works within the SAXO-RT compiler (FTR&D), after the control points have been computed

The communication mechanism is the same as with OCREP: FIFO queues

Technology transfer contract with FTR&D

End of Chapter 2

Chapter 3

Automatic rate desynchronisation of reactive embedded systems

Alain GIRAULT

(Joint work with Paul CASPI, Xavier NICOLLIN,
Daniel PILAUD, and Marc POUZET)

INRIA Grenoble Rhône-Alpes

Introduction

Embedded reactive programs

- embedded so they have limited resources
- reactive so they react continuously with their environment

Introduction

Embedded reactive programs

- embedded so they have limited resources
- reactive so they react continuously with their environment

We consider programs whose control structure is a finite state automaton

Put inside a periodic execution loop:

```
loop each tick
  read inputs
  compute next state
  write outputs
end loop
```

Automatic rate desynchronisation

Desynchronisation: to transform one centralised synchronous program into a GALS program

➪ Each local program is embedded inside its own periodic execution loop

Automatic: the user only provides distribution specifications

Rate desynchronisation:

- the periods of the execution loops will not be the same and
- not necessarily identical to the period of the initial centralised program

Motivation: long duration tasks

Characteristics:

- Their execution time is long
- Their execution time is known and bounded
- Their maximal execution rate is known and bounded

Examples:

- The CO3N4 nuclear plant control system of Schneider Electric
- The Mars rover pathfinder

A small example

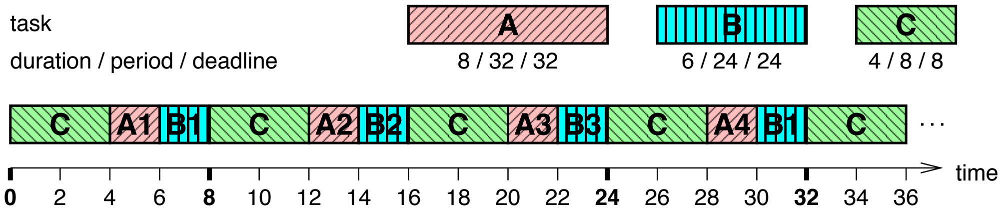
Consider a system with three independant tasks:

- Task A performs slow computations:
  ➪ duration = 8, period = deadline = 32
- Task B performs medium and not urgent computations:
  ➪ duration = 6, period = deadline = 24
- Task C performs fast and urgent computations:
  ➪ duration = 4, period = deadline = 8

How to implement this system?

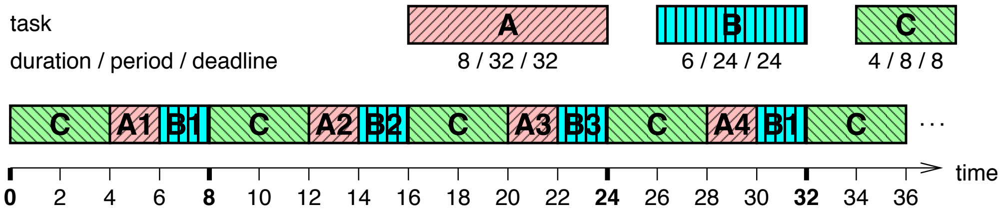
Manual task slicing

Tasks A and B are sliced into small chunks, which are interleaved with task C



Manual task slicing

Tasks A and B are sliced into small chunks, which are interleaved with task C



Very hard and error prone because:

- The slicing is complex
- The implementation must be correct and deadlock-free

Manually programming 3 async. tasks

Tasks A, B, and C are performed by one process each

The task slicing is done by the scheduler of the underlying RTOS

But the manual programming is difficult

Example: the Mars Rover Pathfinder had priority inversion!

Automatic distribution

The user programs a centralised system

The centralised program is compiled, debugged, and validated

It is then automatically distributed into three processes

The correctness ensures that the obtained distributed system is functionnally equivalent to the centralised one

---

Example: the `FILTER` program

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

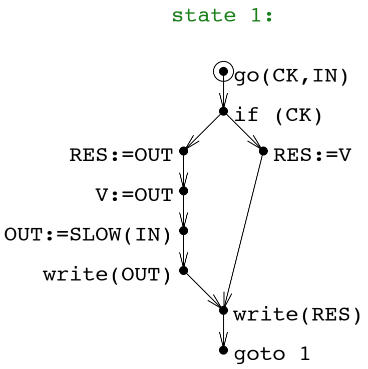
Example: the `FILTER` program

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

Example: the FILTER program

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

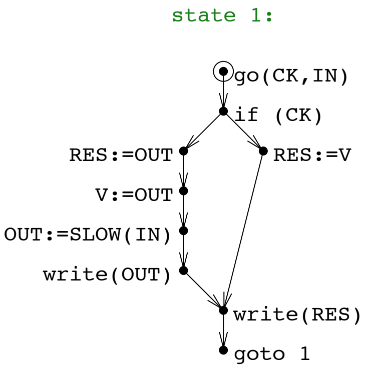
```
state 1:
go(CK,IN)
if (CK) then
    RES:=OUT
    V:=OUT
    OUT:=SLOW(IN)
    write(OUT)
else
    RES:=V
endif
write(RES)
goto 1
```



Example: the FILTER program

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

```
state 1:
go(CK,IN)
if (CK) then
    RES:=OUT
    V:=OUT
    OUT:=SLOW(IN)
    write(OUT)
else
    RES:=V
endif
write(RES)
goto 1
```

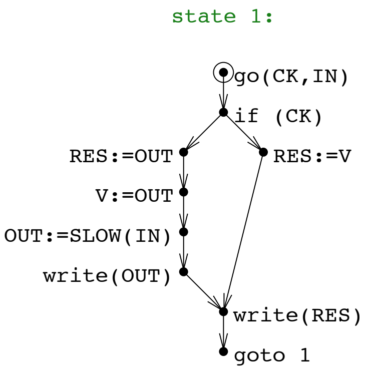


- It has two inputs (the Boolean `CK` and the integer `IN`) and two outputs (the integers `RES` and `OUT`)

Example: the FILTER program

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

```
state 1:
go(CK,IN)
if (CK) then
    RES:=OUT
    V:=OUT
    OUT:=SLOW(IN)
    write(OUT)
else
    RES:=V
endif
write(RES)
goto 1
```



- It has two inputs (the Boolean `CK` and the integer `IN`) and two outputs (the integers `RES` and `OUT`)
- The `go(CK,IN)` action materialises the `read input` phase

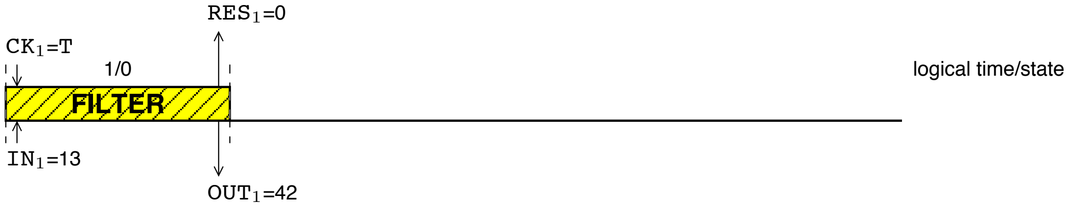
Rates

The `FILTER` program has two inputs (the Boolean `CK` and the integer `IN`) and two outputs (the integers `RES` and `SLOW`)

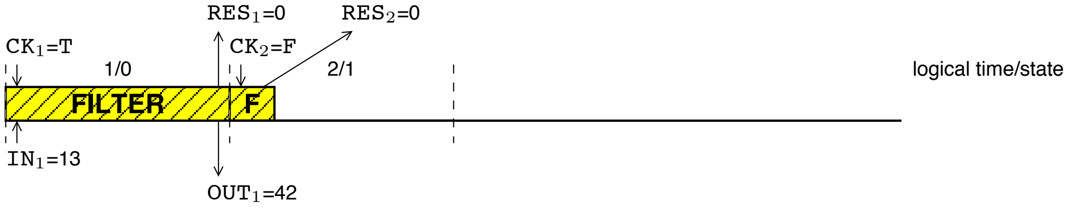
Each input and output has a rate, which is the sequence of logical instants where it exists

- `IN` is used only when `CK` is `true`, so its rate is `CK`
- `CK` is used at each cycle, so its rate is the base rate
- `OUT` is computed each time `CK` is `true`, so its rate is `CK`
- `RES` is computed at each cycle, so its rate is the base rate

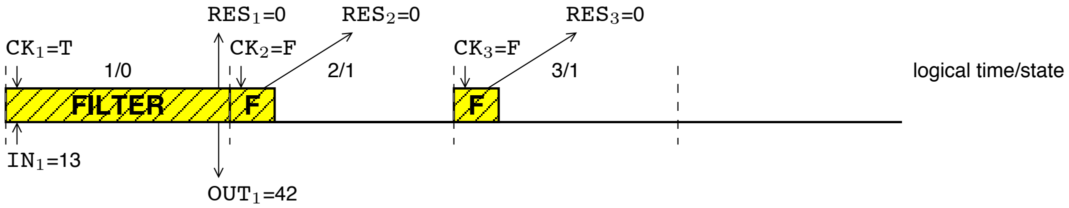
A run of the centralised `FILTER`



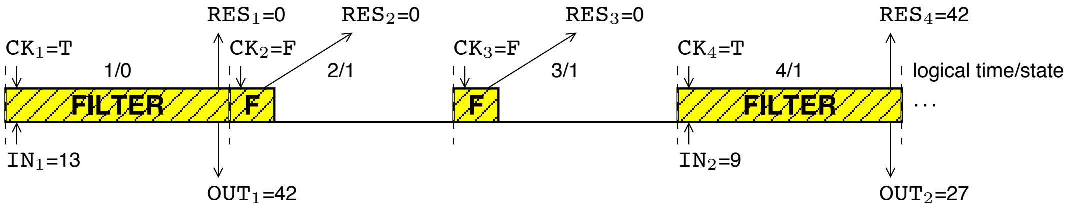
A run of the centralised `FILTER`



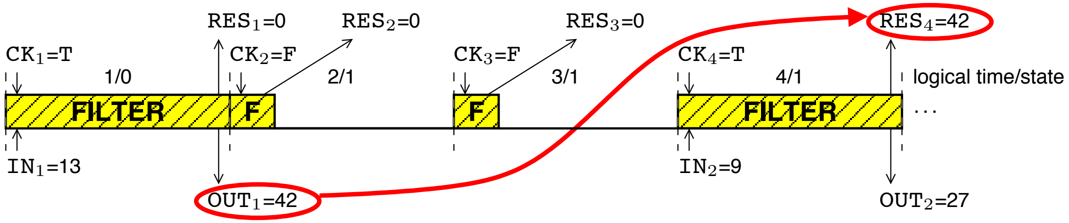
A run of the centralised `FILTER`



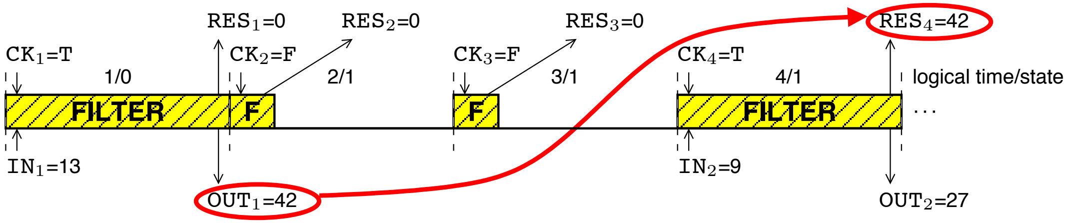
A run of the centralised `FILTER`



A run of the centralised `FILTER`



A run of the centralised `FILTER`

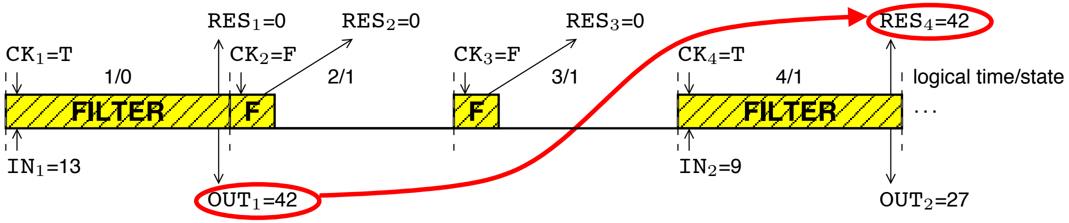


$$\left.\begin{array}{ll}\text{WCET}(\texttt{SLOW}) & = 7 \\ \text{WCET(other computations)} & = 1\end{array}\right\} \Longrightarrow \text{WCET}(\texttt{FILTER}) = 8$$

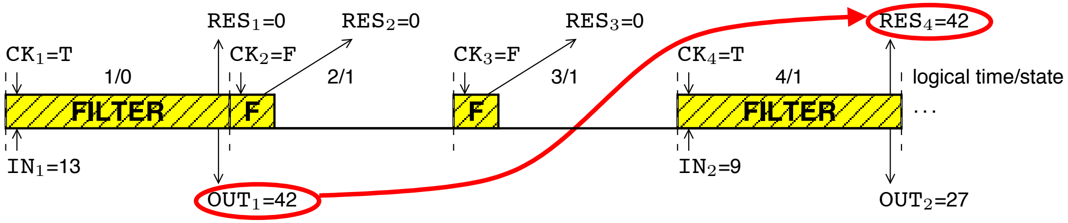
Thus the period of the execution loop (base rate)

must be greater than 8

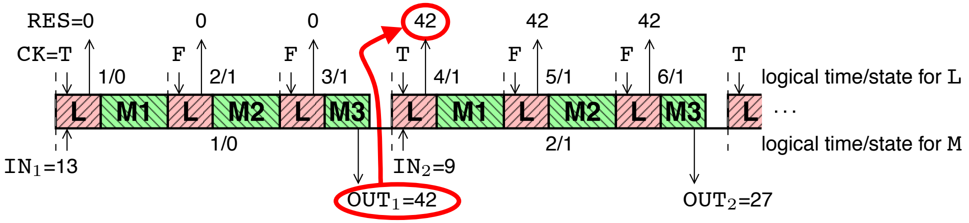
Where are we going?



Where are we going?



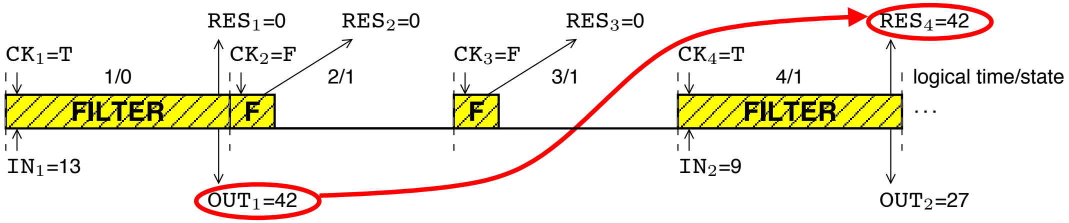
Two tasks running on a single processor:



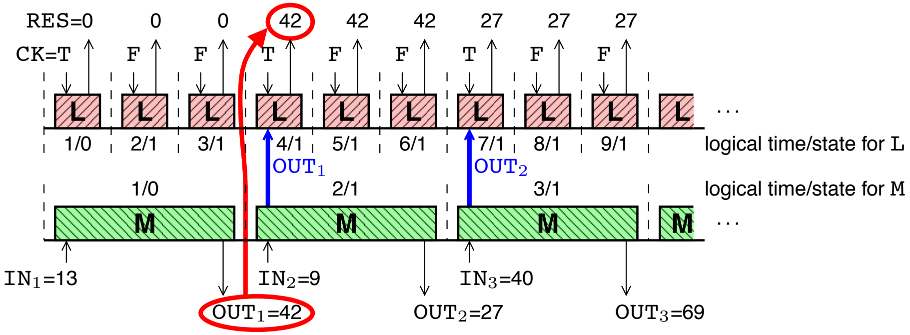
Task `L` performs the fast computations

Task `M` performs the slow computations, sliced into 3 chunks

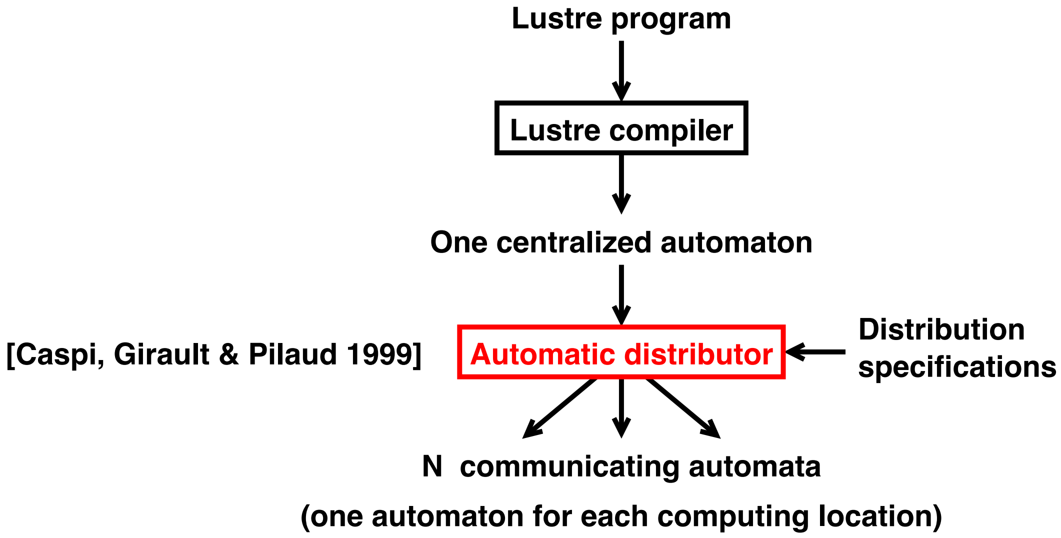
Where are we going?



Two tasks running on two processors:



Our automatic distribution algorithm



Lustre program

Lustre compiler

One centralized automaton

[Caspi, Girault & Pilaud 1999]

Automatic distributor

Distribution specifications

N communicating automata

(one automaton for each computing location)

Communication primitives

Two FIFO channels for each pair of locations, one in each direction:

- `send(dst,var)` inserts the value of variable `var` into the queue directed towards location `dst`

  Non blocking

- `var:=receive(src)` extracts the head value from the queue starting at location `src` and assigns it to variable `var`

  Blocking when the queue is empty

Distribution specifications

location name	assigned rates
L	base
M	CK

This part is given by the user

Distribution specifications

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

The infered inputs and outputs are those whose rate matches the assigned rate

base {RES, CK}
↓
CK {IN, OUT}

Distribution specifications

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

The infered rate is the root of the smallest subtree containing all the rates assigned by the user

---

First attempt of distribution

```
state 0
go(CK,IN)

if (CK) then

   RES:=OUT
   V:=OUT
   OUT:=SLOW(IN)
   write(OUT)
else
   RES:=V
endif
write(RES)
goto 1
```

First attempt of distribution

```
state 0 -- location L        state 0 -- location M
go(CK,IN)                    go(CK,IN)

if (CK) then                 if (CK) then

   RES:=OUT                     RES:=OUT
   V:=OUT                       V:=OUT
   OUT:=SLOW(IN)                OUT:=SLOW(IN)
   write(OUT)                   write(OUT)
else                         else
   RES:=V                       RES:=V
endif                        endif
write(RES)                   write(RES)
goto 1                       goto 1
```

First attempt of distribution

```
state 0 -- location L        state 0 -- location M
go(CK)                       go(IN)

if (CK) then                 if (CK) then

   RES:=OUT
   V:=OUT
                                OUT:=SLOW(IN)
                                write(OUT)
else                         else
   RES:=V

endif                        endif
write(RES)
goto 1                       goto 1
```

First attempt of distribution

```
state 0 -- location L        state 0 -- location M
go(CK)                       go(IN)
send(M,CK)                   CK:=receive(L)
if (CK) then                 if (CK) then
   OUT:=receive(M)              send(L,OUT)
   RES:=OUT
   V:=OUT
                                OUT:=SLOW(IN)
                                write(OUT)
else                         else
   RES:=V

endif                        endif
write(RES)
goto 1                       goto 1
```

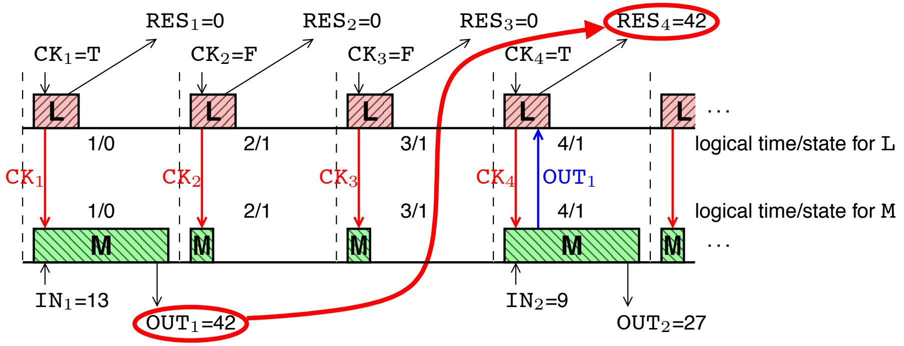
First attempt of distribution

location `L` (rate `base`)   location `M` (rate `CK`)

```
state 0:          state 1:             state 0:            state 1:
go(CK)            go(CK)               go(IN)              go(IN)
send(M,CK)        send(M,CK)           CK:=receive(L)      CK:=receive(L)
if (CK) then {    if (CK) then {       if (CK) then {      if (CK) then {
  RES:=0            OUT:=receive(M)      OUT:=SLOW(IN)       send(L,OUT)
  write(RES)        RES:=OUT             write(OUT)          OUT:=SLOW(IN)
  V:=0              V:=OUT               goto 1              write(OUT)
  goto 1          } else {             } else {            } else {
} else {            RES:=V               goto 0            } endif
  RES:=V          } endif              } endif             goto 1
  write(RES)      write(RES)
  goto 0          goto 1
} endif
```

The `go(CK,IN)` has been split into
- `go(CK)` on location `L`
- `go(IN)` on location `M`

A run of the distributed `FILTER`



The value of `CK` is sent by `L` to `M` at each cycle of the base rate

➪ location `M` runs at the speed of the base rate instead of `CK`

If the communications take 1, then the global WCET is still 8

How to improve this?

We want location `M` to run at the speed of `CK`

➪ This would give enough time for the computation of `SLOW`

➪ For this, location `L` must not send `CK` to location `M`

- We can use an existing bisimulation for detecting and suppressing branchings like `if(CK)` on location `M`
- For this bisimulation to work, the `go(IN)` action must be moved inside the `then` branch on location `M`

  Makes sense because `IN` is expected only when `CK` is `true`

➪ The two programs will be logically desynchronized

Moving the go downward

Only the locations whose rate is not the base rate

A simple forward traversal of the program:

Moving the go downward

Only the locations whose rate is not the base rate

A simple forward traversal of the program:

```
loc. M (rate CK) - state 0
go(IN)
if (CK) then
   OUT:=SLOW(IN)
   write(OUT)
   goto 1
else
   goto 0
endif
```

Moving the go downward

Only the locations whose rate is not the base rate

A simple forward traversal of the program:

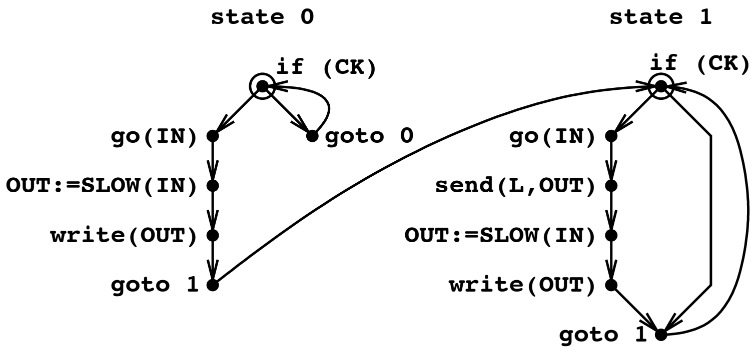
```
loc. M (rate CK) - state 0
go(IN)
if (CK) then
   OUT:=SLOW(IN)
   write(OUT)
   goto 1
else
   goto 0
endif
```

⇝

```
loc. M (rate CK) - state 0
if (CK) then
   go(IN)
   OUT:=SLOW(IN)
   write(OUT)
   goto 1
else
   goto 0
endif
```

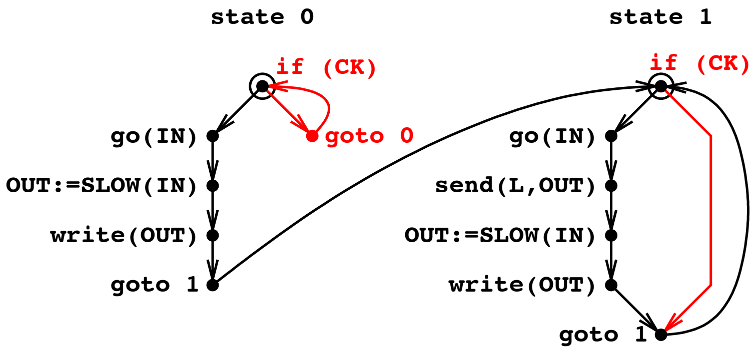
Suppressing useless branchings

Bisimulation fully presented in [Caspi, Fernandez & Girault 1995]



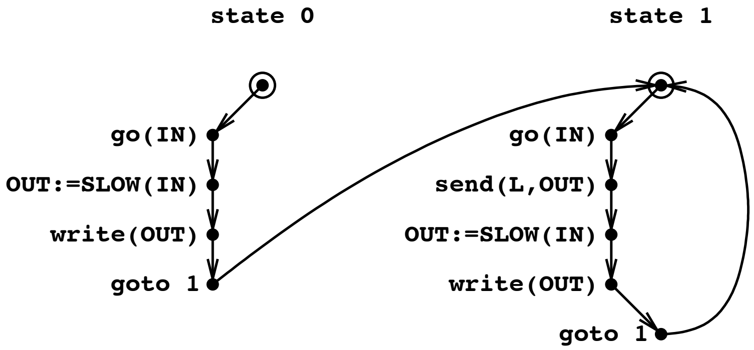
Suppressing useless branchings

Bisimulation fully presented in [Caspi, Fernandez & Girault 1995]



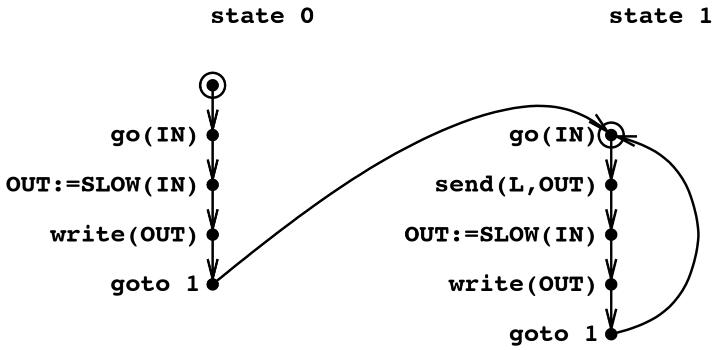
Suppressing useless branchings

Bisimulation fully presented in [Caspi, Fernandez & Girault 1995]



Suppressing useless branchings

Bisimulation fully presented in [Caspi, Fernandez & Girault 1995]



Final result

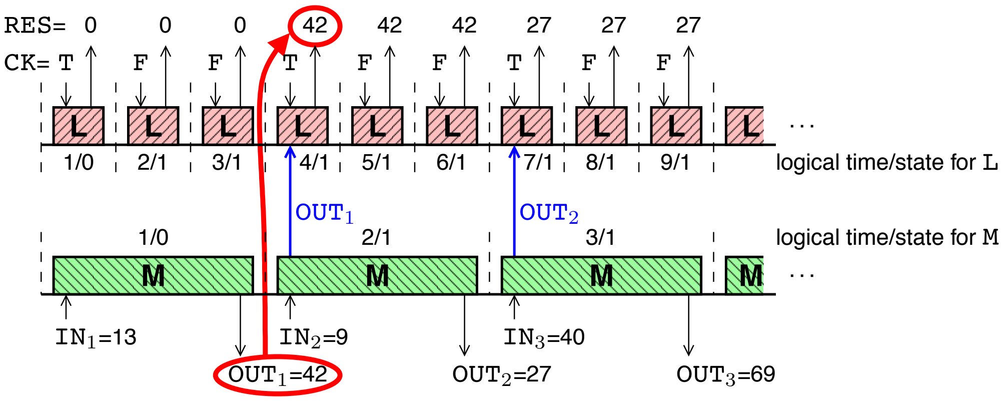
location `L` (rate `base`)

```
state 0:               state 1:
go(CK)                 go(CK)
if (CK) then {         if (CK) then {
  RES:=0                 OUT:=receive(M)
  write(RES)             RES:=OUT
  V:=0                   V:=OUT
  goto 1               } else {
} else {                 RES:=V
  RES:=V               } endif
  write(RES)           write(RES)
  goto 0               goto 1
} endif
```

location `M` (rate `CK`)

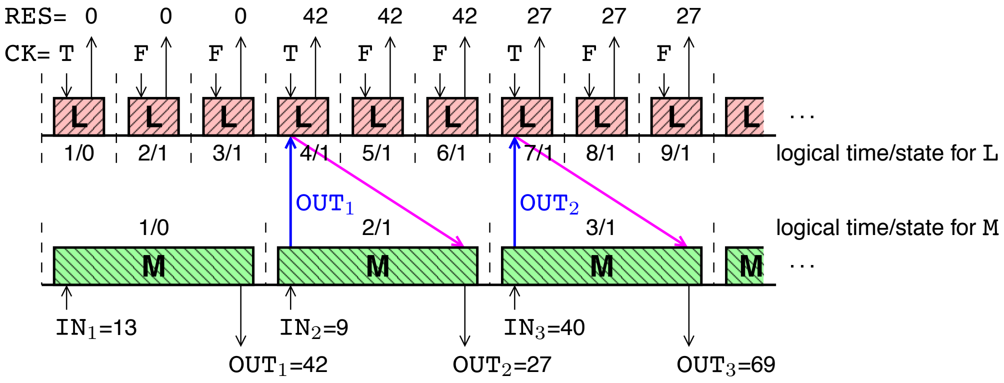
```
state 0:               state 1:
go(IN)                 go(IN)
OUT:=SLOW(IN)          send(L,OUT)
write(OUT)             OUT:=SLOW(IN)
goto 1                 write(OUT)
                       goto 1
```

A run of the newly distributed **`FILTER`**



The period of `L` is one third of the period of `M`

A run of the newly distributed **`FILTER`**



Dummy communications can finally be added to guarantee bounded FIFO queues

---

Validating the synchronous abstraction

We have to compare the WCET with the execution loop period

But our program is distributed into n tasks. So:

- ➪ We compute the n WCET
- ➪ We compute the total utilisation factor
- ➪ We check the Liu & Layland conditions (mono-processor case)

Validating the synchronous abstraction

We have to compare the WCET with the execution loop period

But our program is distributed into n tasks. So:

- We compute the n WCET
- We compute the total utilisation factor
- We check the Liu & Layland conditions (mono-processor case)

location	`L`	`M`
WCET	2	8
rate	5	15

Validating the synchronous abstraction

We have to compare the WCET with the execution loop period

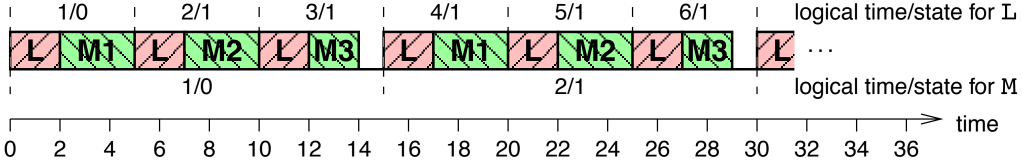
But our program is distributed into n tasks. So:

- We compute the n WCET
- We compute the total utilisation factor
- We check the Liu & Layland conditions (mono-processor case)

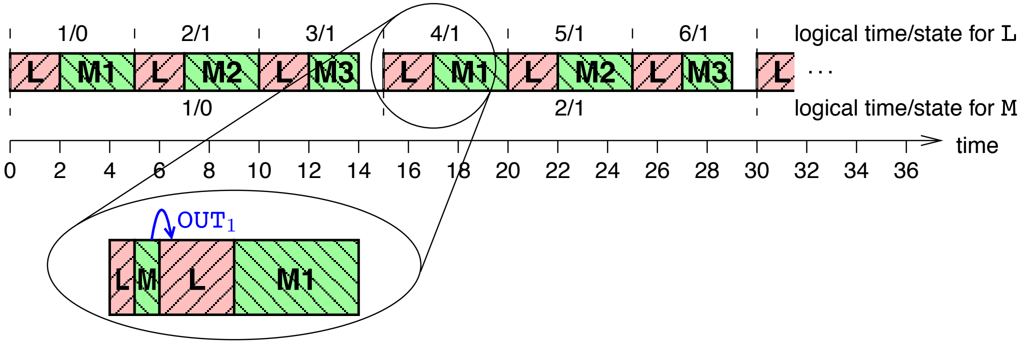
location	`L`	`M`
WCET	2	8
rate	5	15

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

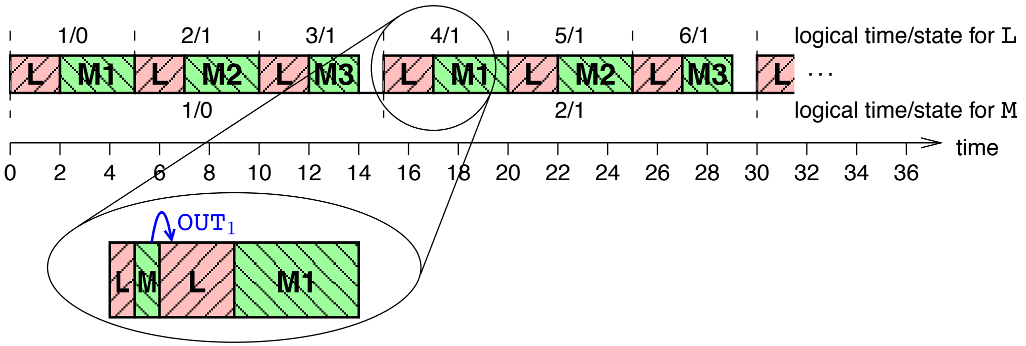
RTOS implementation



RTOS implementation

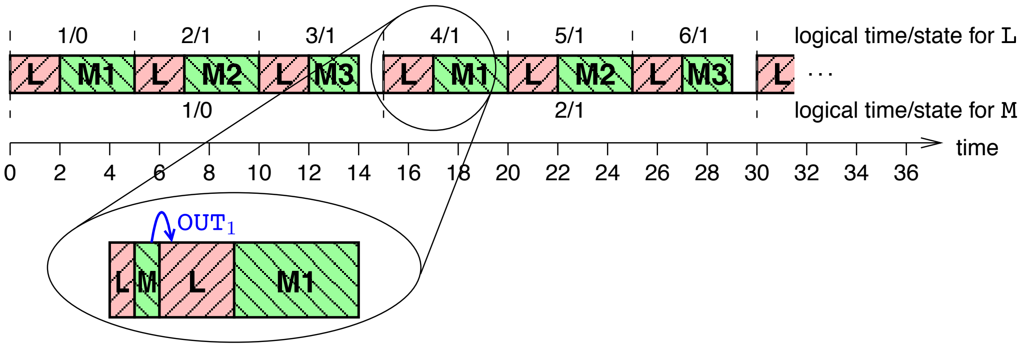


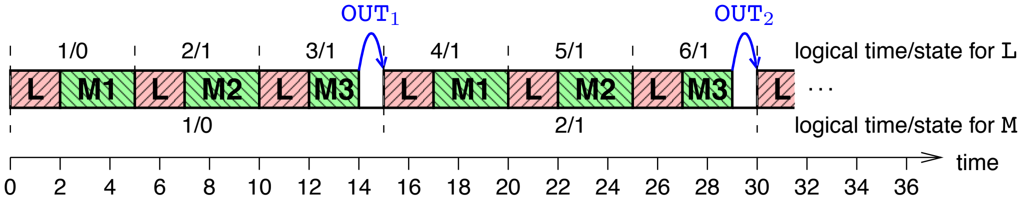
RTOS implementation



This mechanism relies on the preemption mechanism of the RTOS!

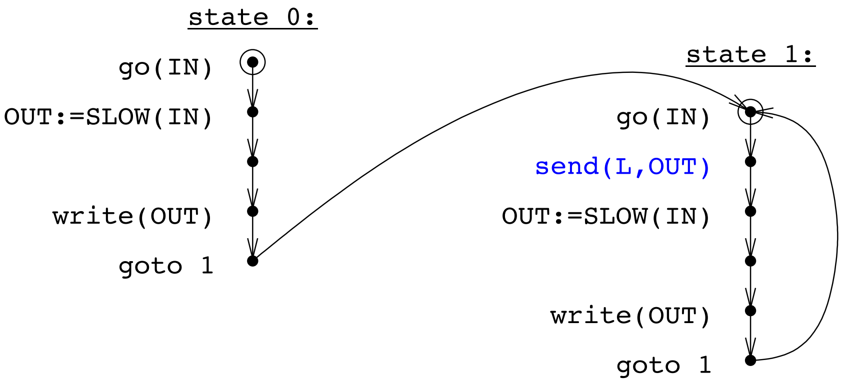
RTOS implementation





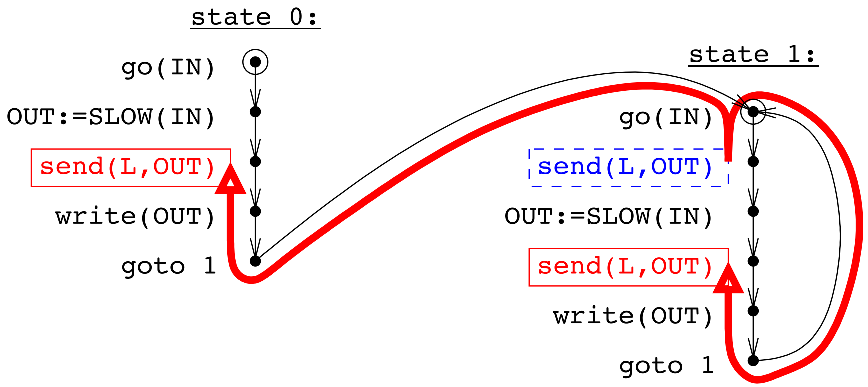
Data-flow analysis

Program of location `M`



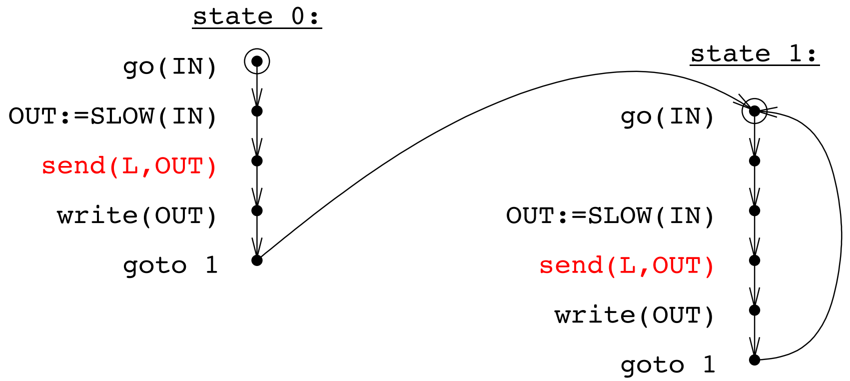
Data-flow analysis

Program of location `M`



Data-flow analysis

Program of location `M`



Two applications

1. Clock driven automatic distribution of Lustre programs
2. Automatic rate desynchronisation of Esterel programs

Lustre is synchronous, declarative, data-flow

All objects are flows: infinite sequences of typed data

Clocks

Each flow has a clock (= *first class abstract type*)

➪ The sequence of instants where the flow bears a value

Any Boolean flow defines a new clock: the sequence of instants where it bears the value `true`

Flows can then be upsampled (`current`)
and downsampled (`when`)

A program must be correctly clocked

One clock is called the base clock of the program:
➪ the sequence of its activation instants (the Esterel `tick`)

The set of clocks is a tree whose root is the base clock

Syntax

```
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);
```

Syntax

```
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);
```

The `SLOW` function is long duration task

Syntax

```
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);
```

The clock tree is:

base {`RES`, `CK`}
↓
`CK` {`IN`, `OUT`}

An example of a run of **`FILTER`**

base clock 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			...

An example of a run of **`FILTER`**

base clock 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 = SLOW(IN)`	42			27			69			...

An example of a run of `FILTER`

base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...

An example of a run of `FILTER`

base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...

An example of a run of `FILTER`

base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...
`(0 when CK) -> pre OUT`	0			42			27			...

An example of a run of `FILTER`

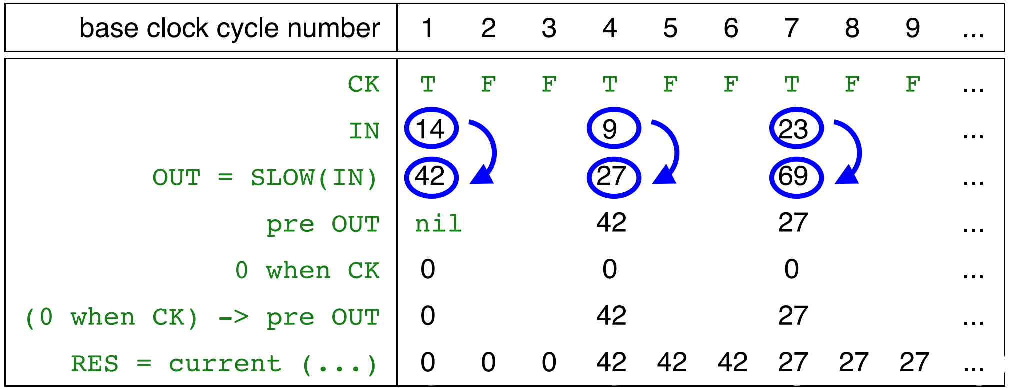
base clock cycle number	1	2	3	4	5	6	7	8	9	...
`CK`	`T`	`F`	`F`	`T`	`F`	`F`	`T`	`F`	`F`	...
`IN`	14			9			23			...
`OUT = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...
`(0 when CK) -> pre OUT`	0			42			27			...
`RES = current (...)`	0	0	0	42	42	42	27	27	27	...

An example of a run of `FILTER`

base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...
`(0 when CK) -> pre OUT`	0			42			27			...
`RES = current (...)`	0	0	0	42	42	42	27	27	27	...

- These are logical instants

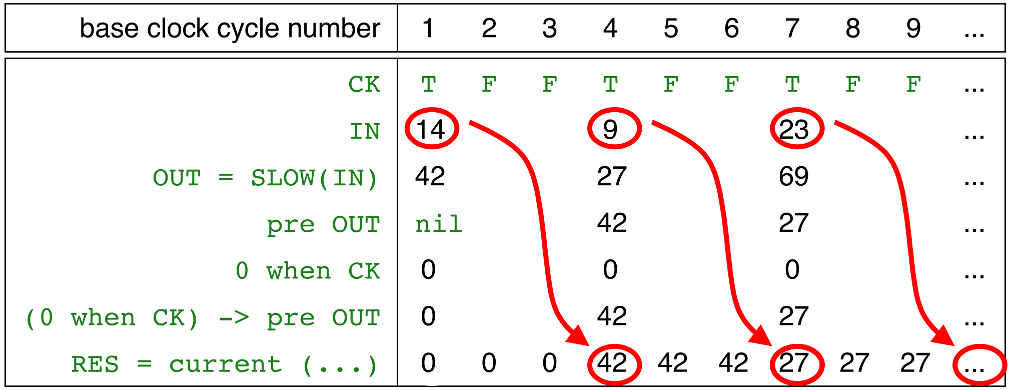
An example of a run of `FILTER`



base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...
`(0 when CK) -> pre OUT`	0			42			27			...
`RES = current (...)`	0	0	0	42	42	42	27	27	27	...

- These are logical instants
- `OUT` must be available at the same clock cycle of `CK` as `IN`

An example of a run of `FILTER`



base clock 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 = SLOW(IN)`	42			27			69			...
`pre OUT`	`nil`			42			27			...
`0 when CK`	0			0			0			...
`(0 when CK) -> pre OUT`	0			42			27			...
`RES = current (...)`	0	0	0	42	42	42	27	27	27	...

- These are logical instants
- `OUT` must be available at the same clock cycle of `CK` as `IN`
- `RES` must be available at the next clock cycle of `CK`

Clock-driven automatic distribution

Automatic distribution:

From a centralised source program and some distribution specifications, we build automatically as many programs as required by the user

Their combined behaviour will be functionnaly equivalent to the behaviour of the initial centralised program

Clock-driven automatic distribution

Automatic distribution:

From a centralised source program and some distribution specifications, we build automatically as many programs as required by the user

Their combined behaviour will be functionnaly equivalent to the behaviour of the initial centralised program

Clock-driven:

The user specifies which clock goes to which computing location

➪ Partition of the set of clocks of the centralised source program

One subset for each desired computing location

Related work

- Giotto compiler: [Henzinger, Horowitz & Kirsch 2001]
- Asynchronous tasks in Esterel: [Paris 1992]
- Automatic distribution in Signal: [Maffeis 1993],
  [Aubry, Le Guernic, Machard 1996],
  [Benveniste, Caillaud & Le Guernic 2000]
- Distributed implementation of Lustre over TTA:
  [Caspi, Curic, Maignan, Sofronis, Tripakis & Niebert 2003]
- Futures in Heptagon: [Gérard 2013]

Asynchronous tasks in Esterel

Tasks are external computation entities syntactically similar to procedures, but the execution of which is assumed to be non-instantaneous.

```
module FILTER:
  input CK;
  input IN : integer;
  output RES, OUT : integer;
  task SLOW(integer)(integer);
  return R;

  loop
    present CK then
      exec SLOW(OUT)(IN) return R;
    else
      emit RES (pre(?RES))
    end present
    ||
    present R then
      RES = ?OUT;
    end present
  each tick
end module
```

Futures in Heptagon

A future is a computation the evaluation of which is launched concurrently, and the result of which is expected later.

```
node SLOW (A:int) returns (B:int)
let
  do some computations();
tel

node FILTER (CK:bool, IN:int) returns (RES:int)
var OUT : future int;
let
  OUT = async SLOW (IN);
  RES = merge CK (!((async 0) fby OUT))
                 (0 fby (RES whenot CK));
tel
```

End of chapter 3