

Automatic rate desynchronisation of reactive embedded systems

Paul CASPI, Alain GIRAULT, Xavier NICOLLIN, Daniel PILAUD, and
Marc POUZET

INRIA Rhône-Alpes, INPG-VERIMAG, and Orsay/LRI

Grenoble and Paris, FRANCE

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

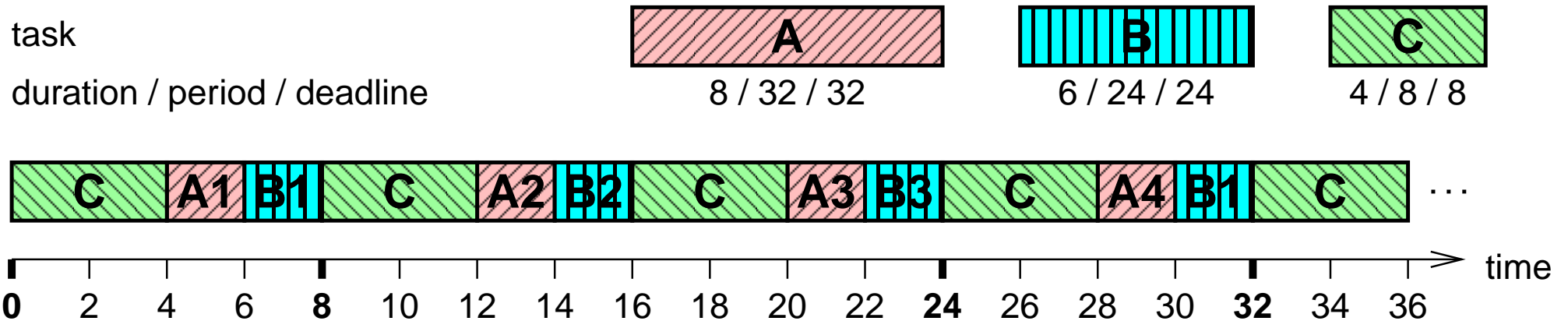
Consider a system with three independent tasks:

- 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?

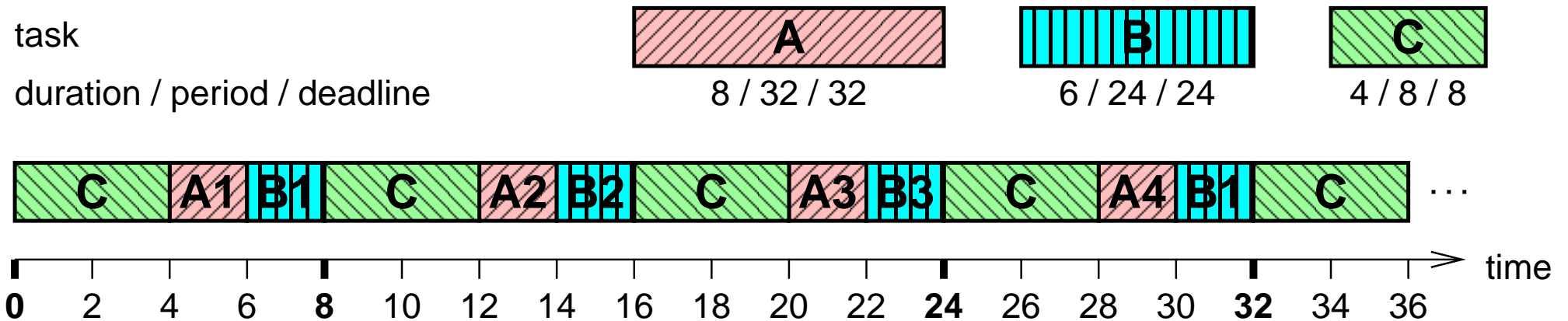
Manual task slicing

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:
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

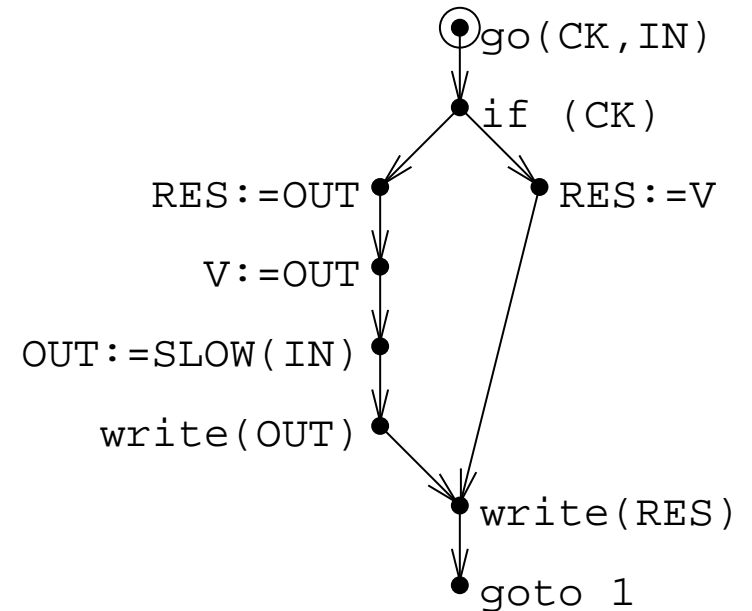
state 0:

```
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
```

state 1:



Example: the **FILTER** program

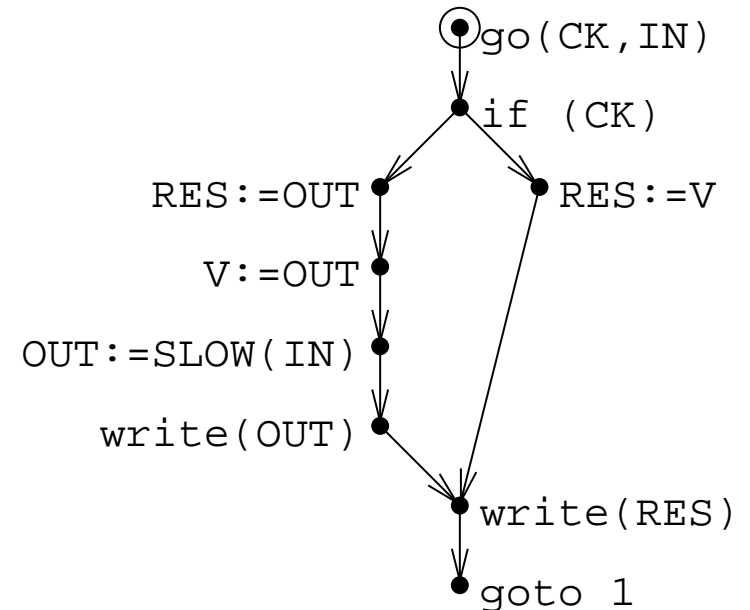
state 0:

```
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
```

state 1:



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

Example: the **FILTER** program

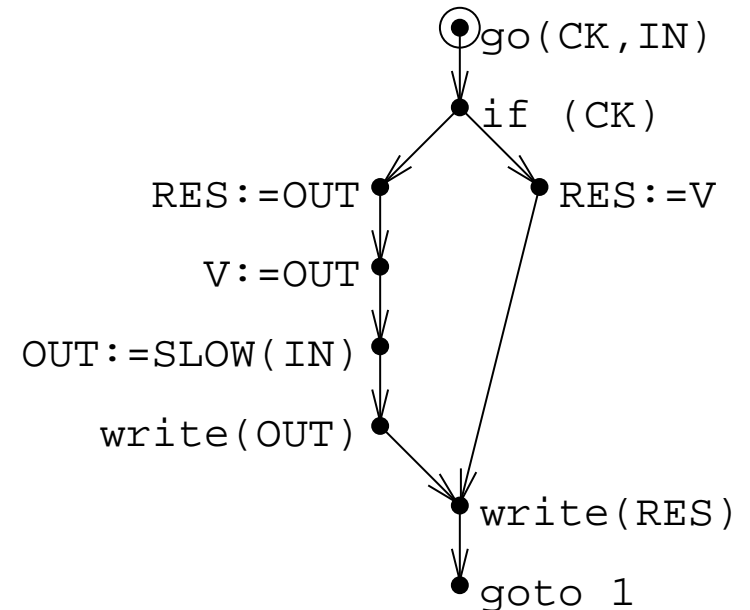
state 0:

```
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
```

state 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

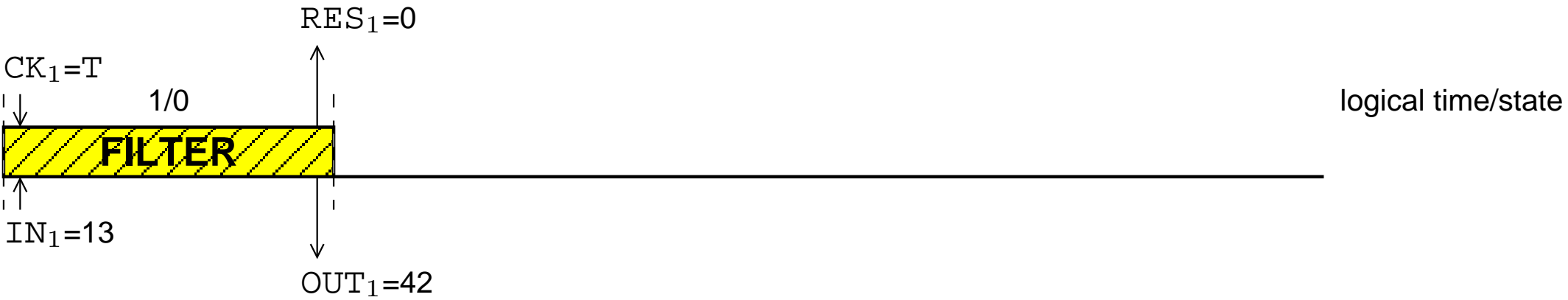
Rates

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

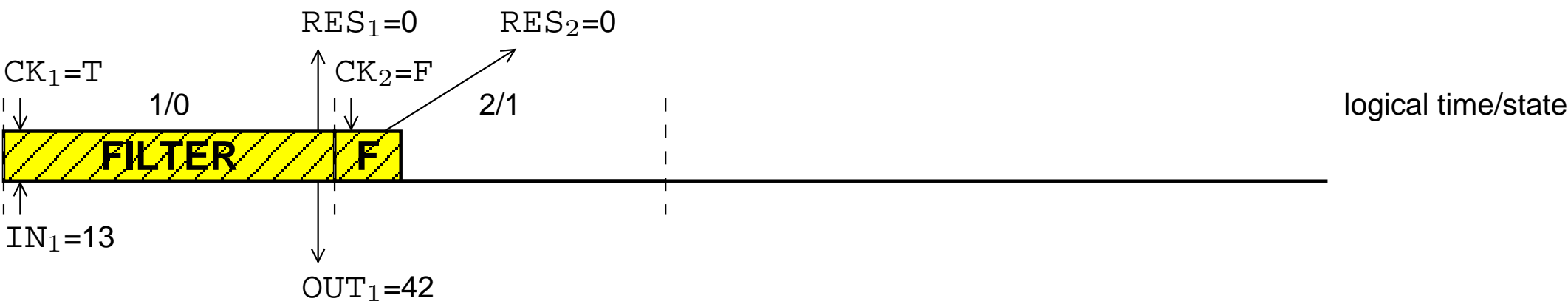
Each input and output has a **rate**, which is the sequence of logical instants where it exists

- `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

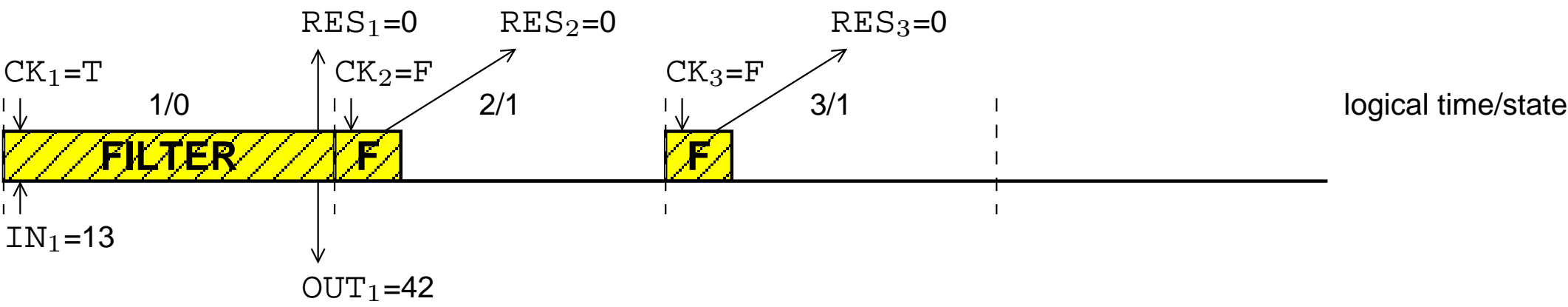
A run of the centralised FILTER



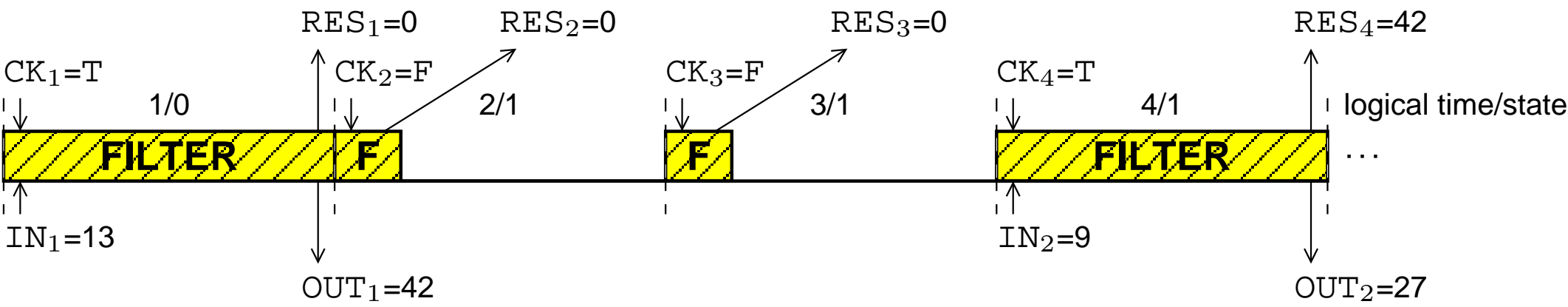
A run of the centralised FILTER



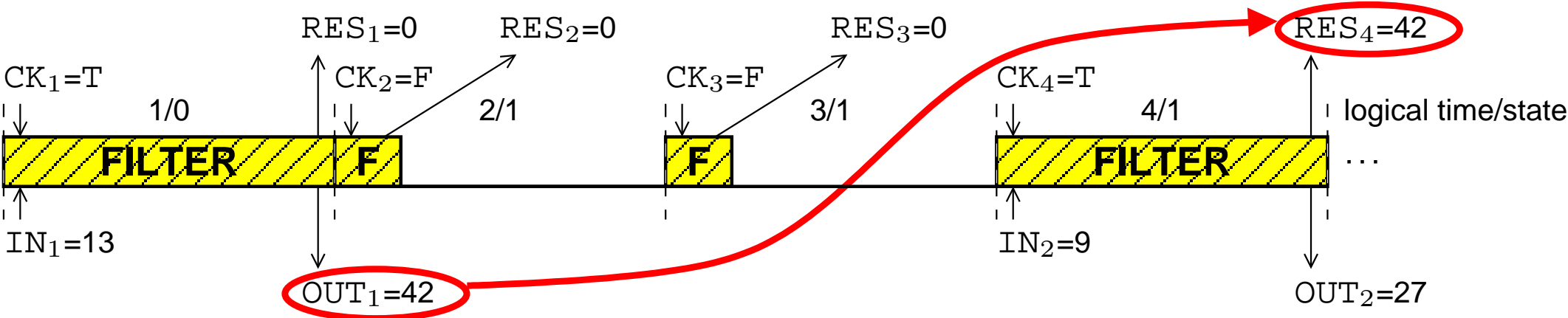
A run of the centralised FILTER



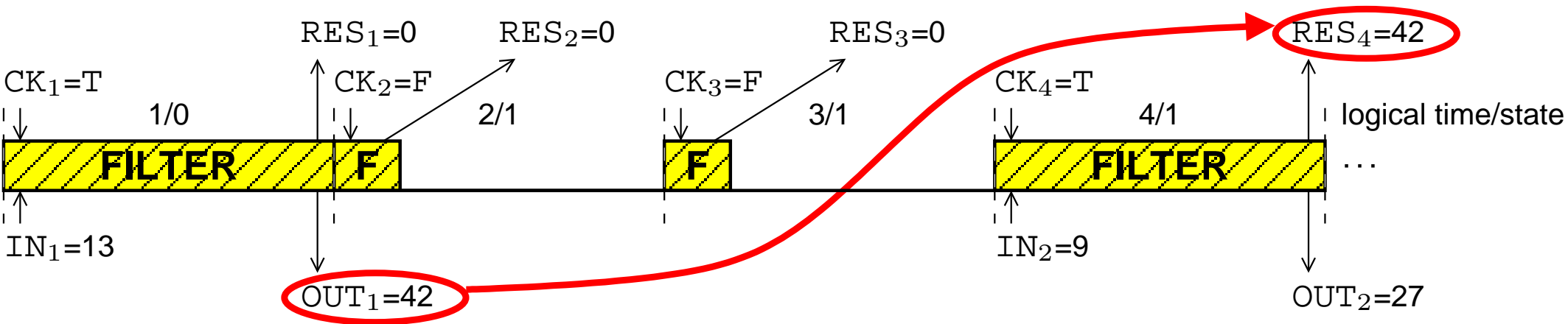
A run of the centralised FILTER



A run of the centralised FILTER



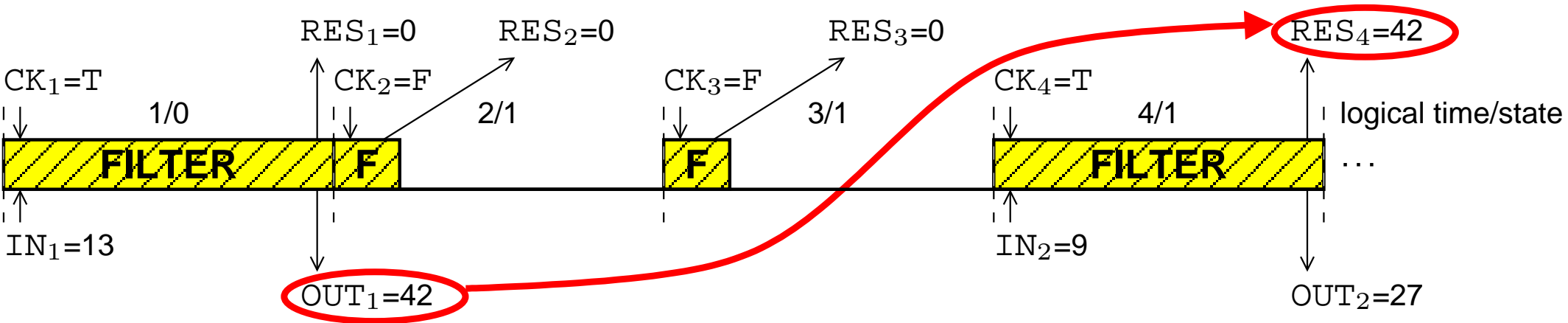
A run of the centralised FILTER



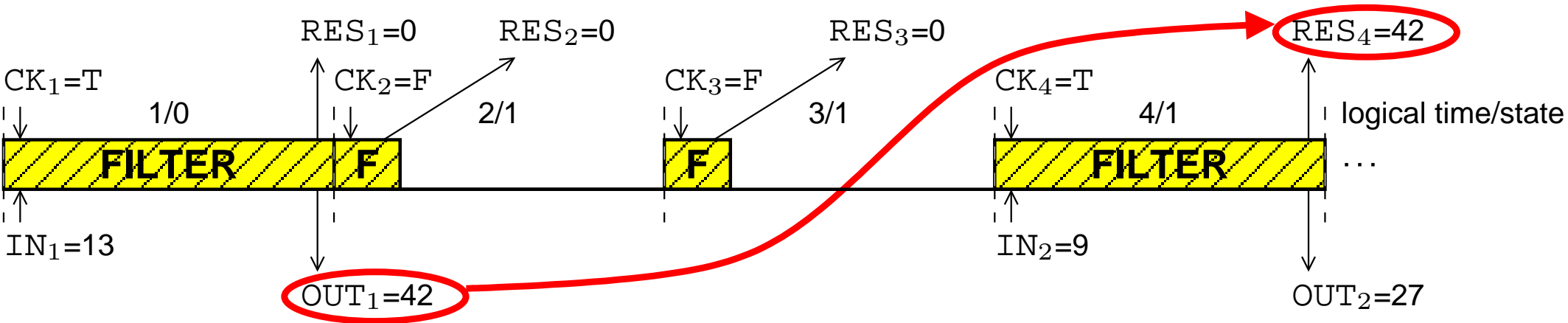
$$\left. \begin{array}{l} \text{WCET}(\text{SLOW}) = 7 \\ \text{WCET}(\text{other computations}) = 1 \end{array} \right\} \implies \text{WCET}(\text{FILTER}) = 8$$

Thus the period of the execution loop (base rate)
must be **greater than 8**

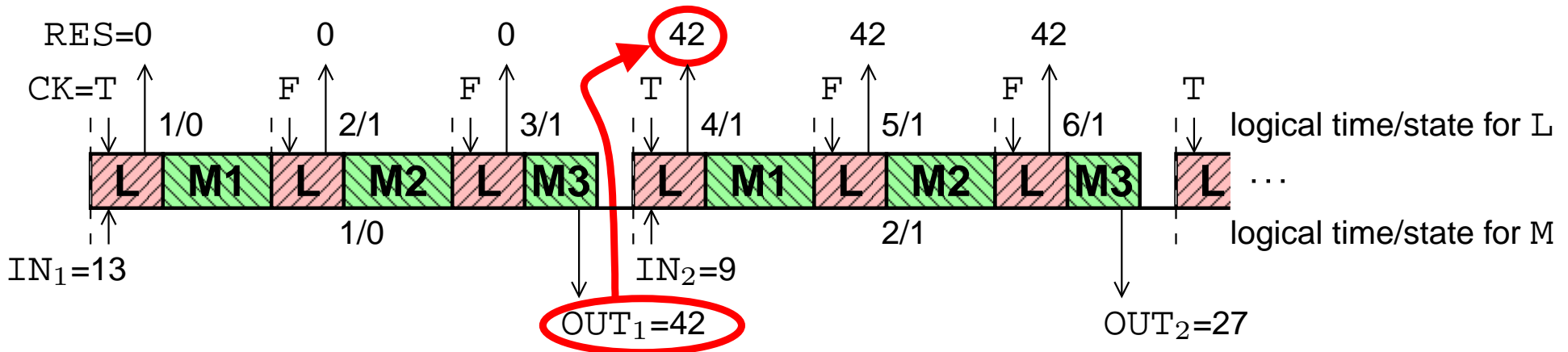
Where are we going?



Where are we going?



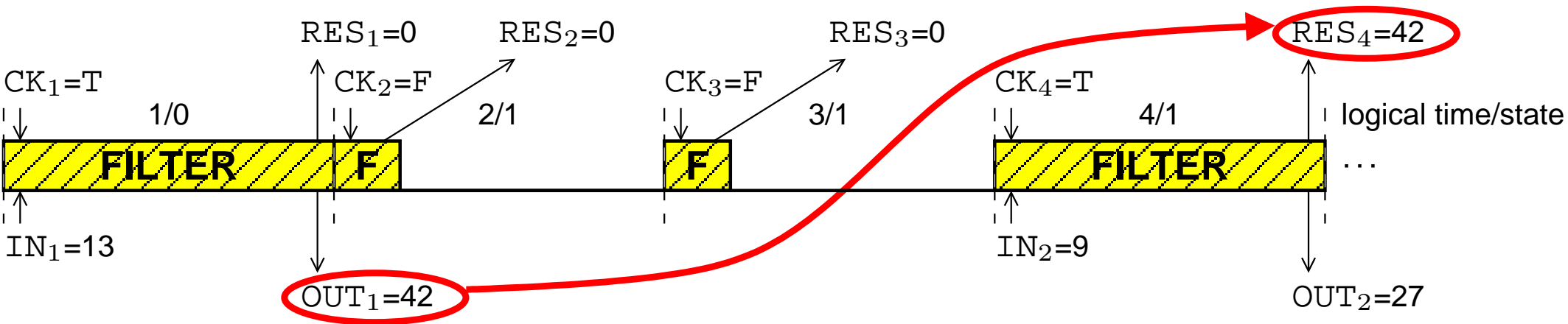
Two tasks running on a **single** processor:



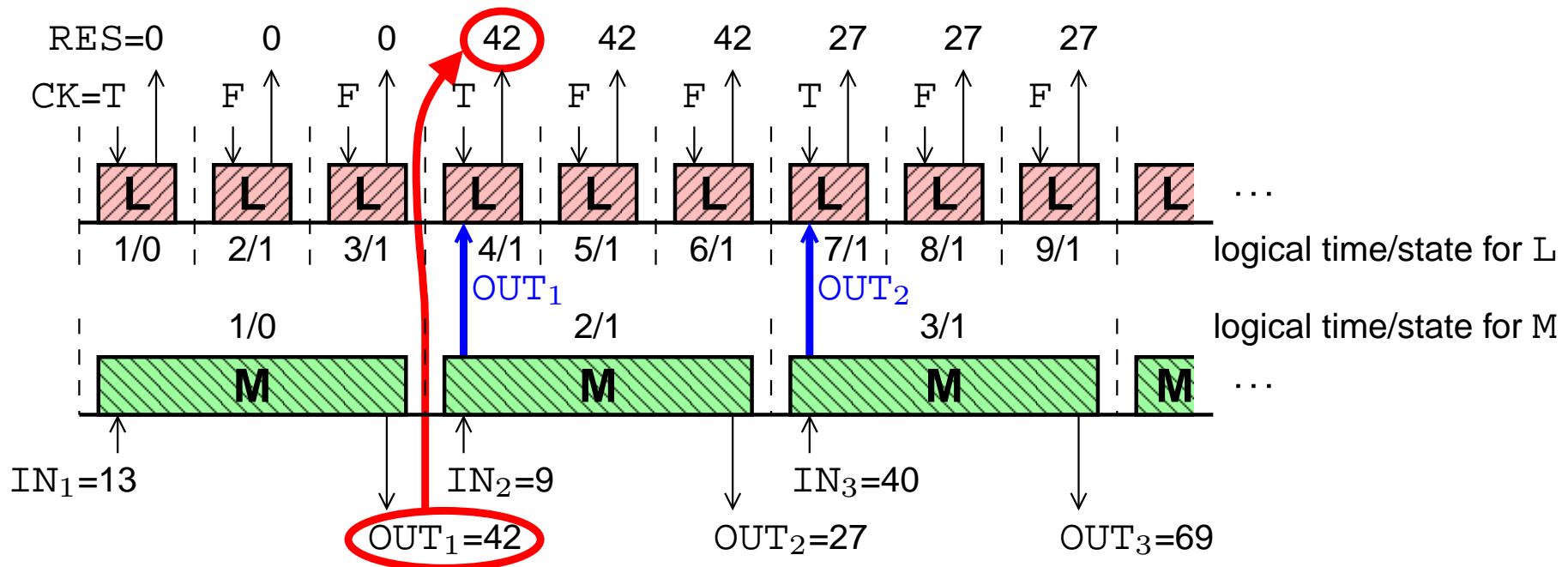
Task **L** performs the **fast** computations

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

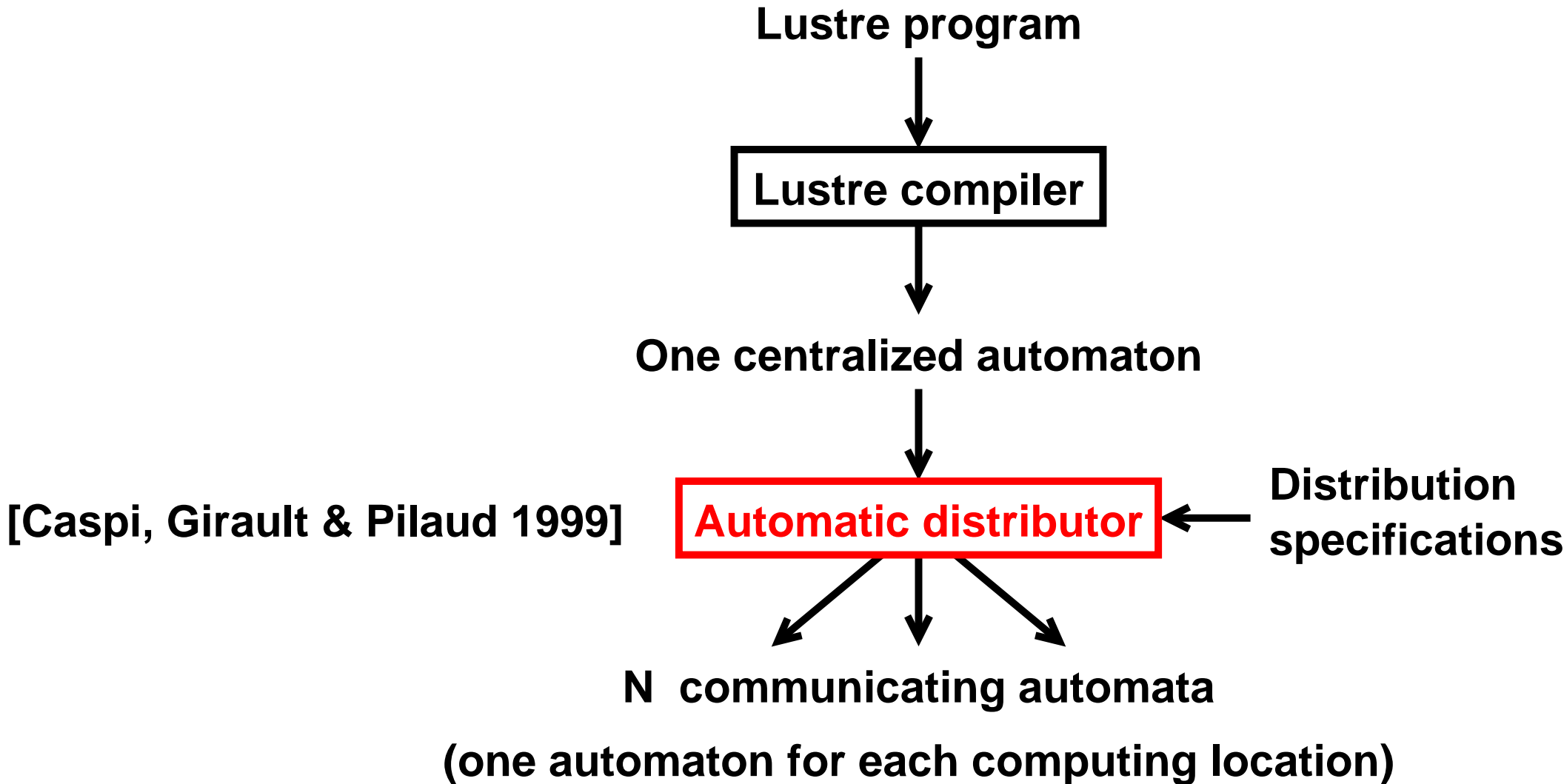
Where are we going?



Two tasks running on two processors:



Our automatic distribution algorithm



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	inferred inputs & outputs
L	base	CK, RES
M	CK	IN, OUT

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

base {RES, CK}
↓
CK {IN, OUT}

Distribution specifications

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

The inferred 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

go(CK, IN)

if (CK) then

RES:=OUT

V:=OUT

OUT:=SLOW(IN)

write(OUT)

else

RES:=V

endif

write(RES)

goto 1

state 0 -- location M

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

go(CK)

if (CK) then

RES:=OUT

V:=OUT

else

RES:=V

endif

write(RES)

goto 1

state 0 -- location M

go(IN)

if (CK) then

OUT:=SLOW(IN)

write(OUT)

else

endif

goto 1

First attempt of distribution

state 0 -- location L

go(CK)

send(M,CK)

if (CK) then

 OUT:=receive(M)

 RES:=OUT

 V:=OUT

else

 RES:=V

endif

write(RES)

goto 1

state 0 -- location M

go(IN)

CK:=receive(L)

if (CK) then

 send(L,OUT)

 OUT:=SLOW(IN)

 write(OUT)

else

endif

goto 1

First attempt of distribution

location L (rate base)

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

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

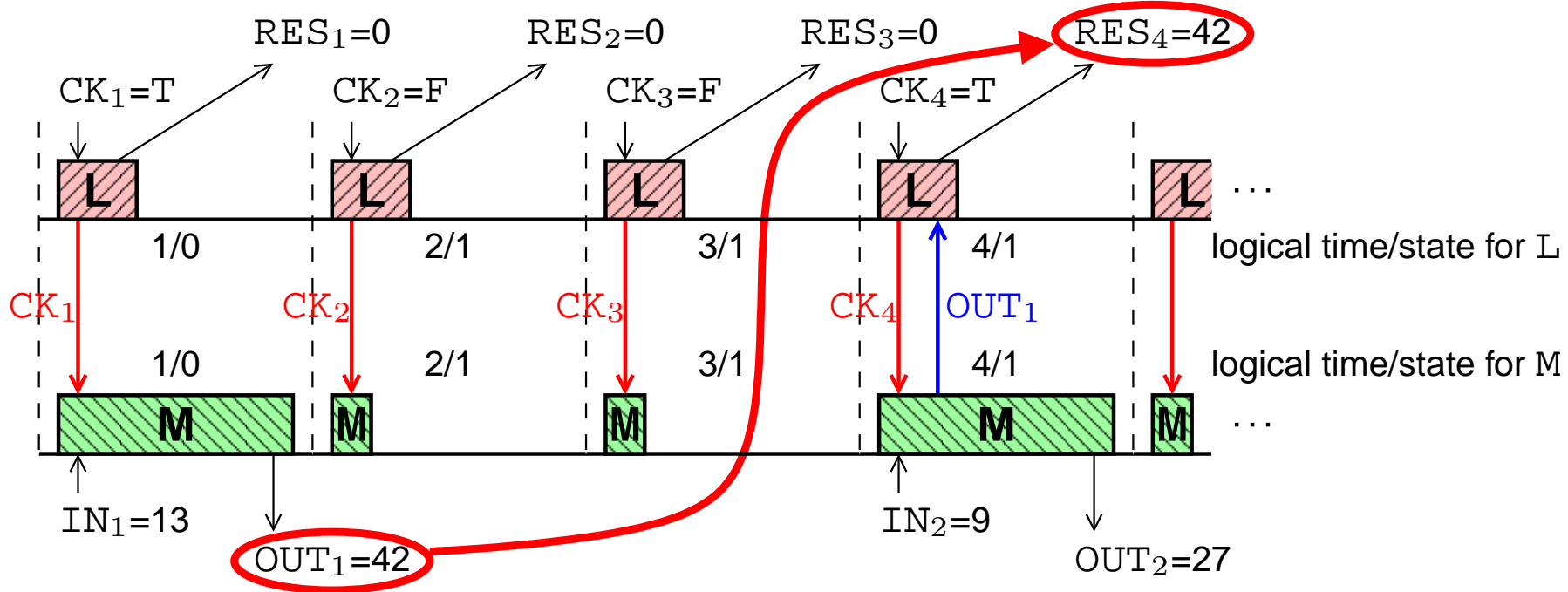
location M (rate CK)

```
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
```

The `go(CK, IN)` has been split into $\left\{ \begin{array}{l} \text{go(CK) on location L} \\ \text{go(IN) on location M} \end{array} \right.$

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

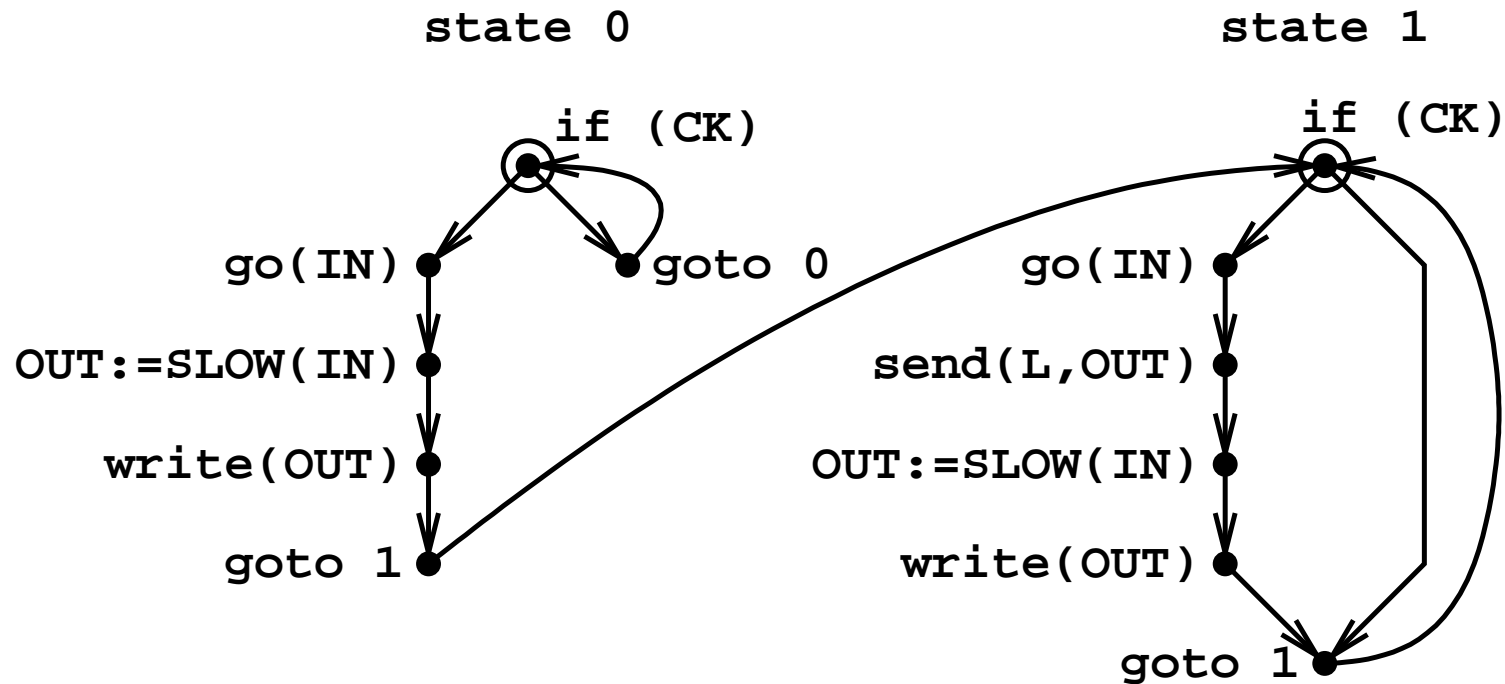
Only the locations whose rate **is not** the base rate

A simple forward traversal of the program:

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

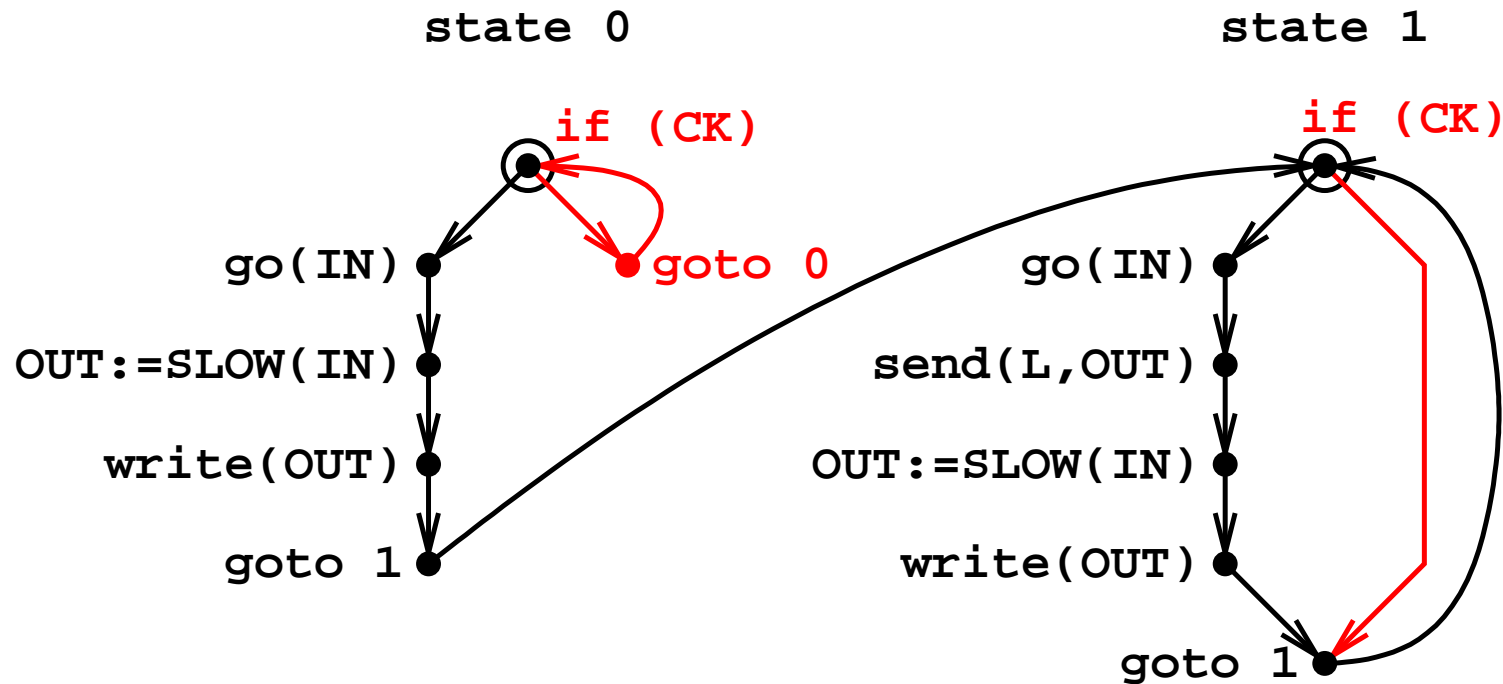
Suppressing useless branchings

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



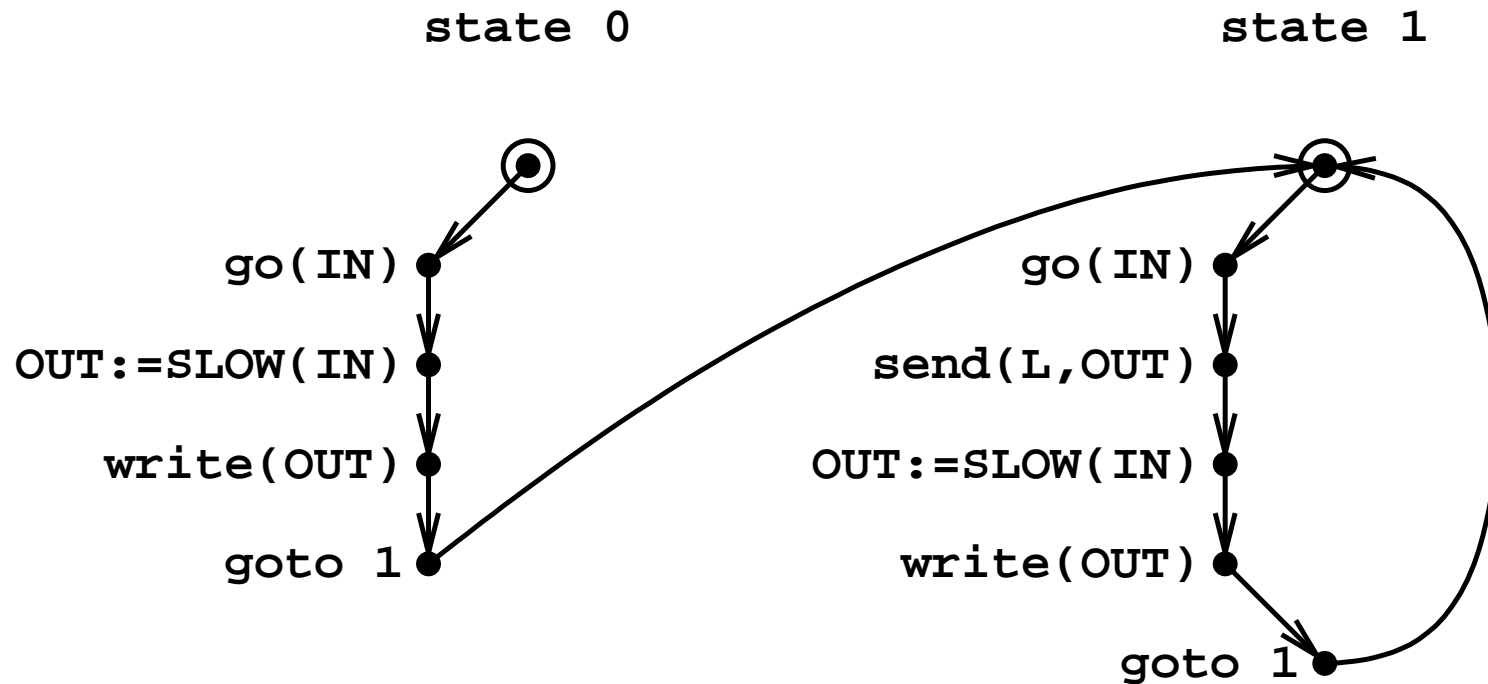
Suppressing useless branchings

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



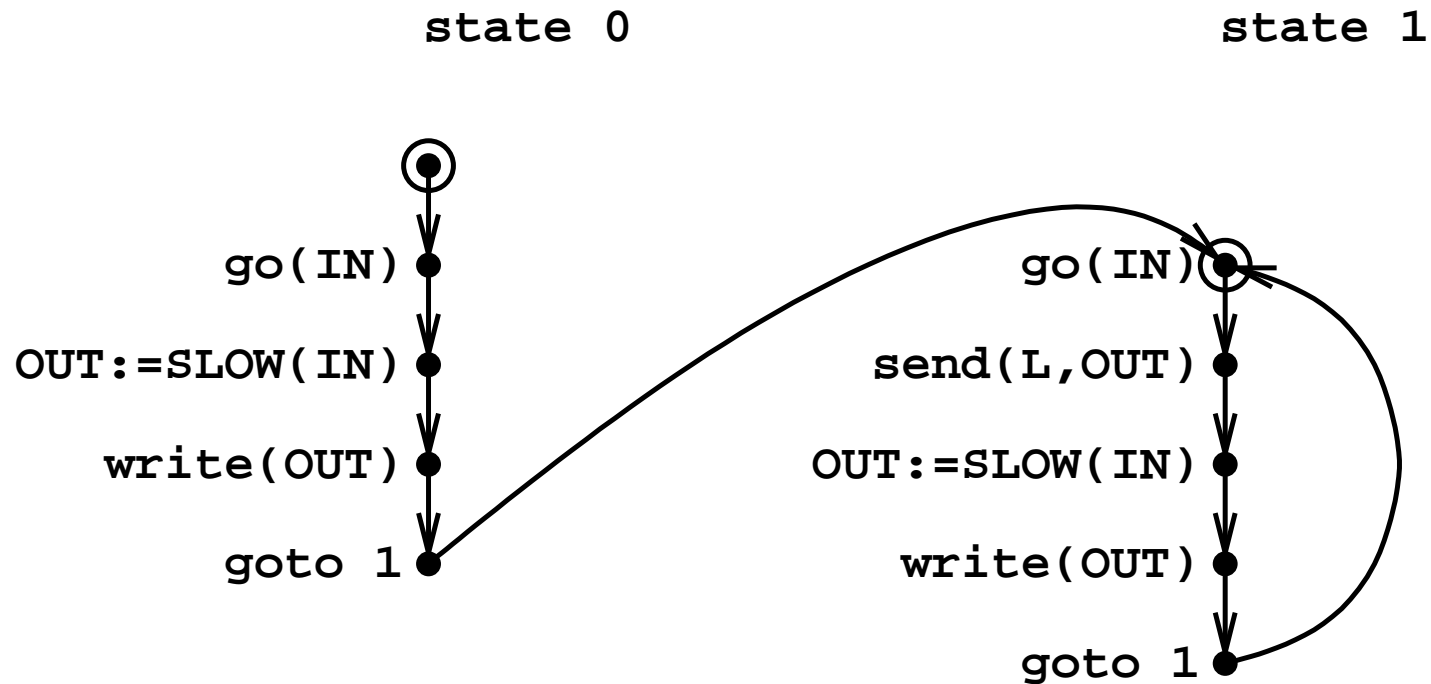
Suppressing useless branchings

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

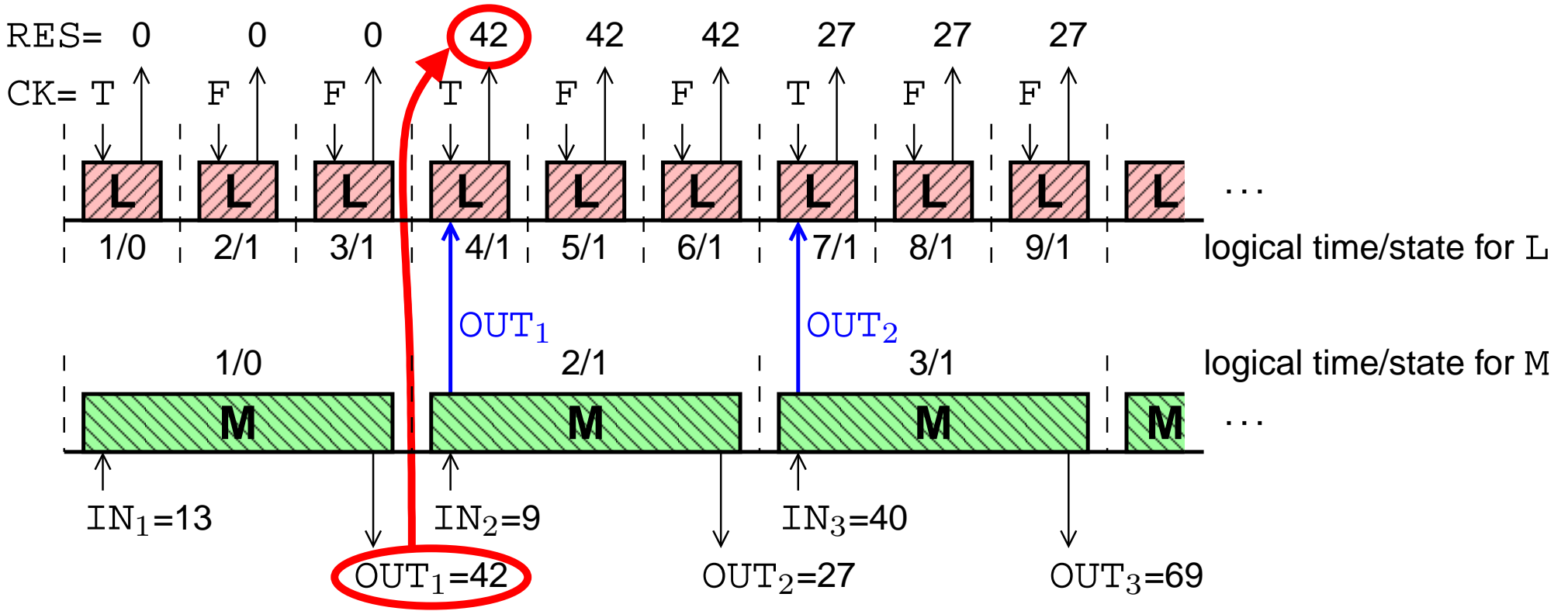
state 1:
go(CK)
if (CK) then {
  OUT:=receive(M)
  RES:=OUT
  V:=OUT
} else {
  RES:=V
} endif
write(RES)
goto 1
```

location M (rate CK)

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

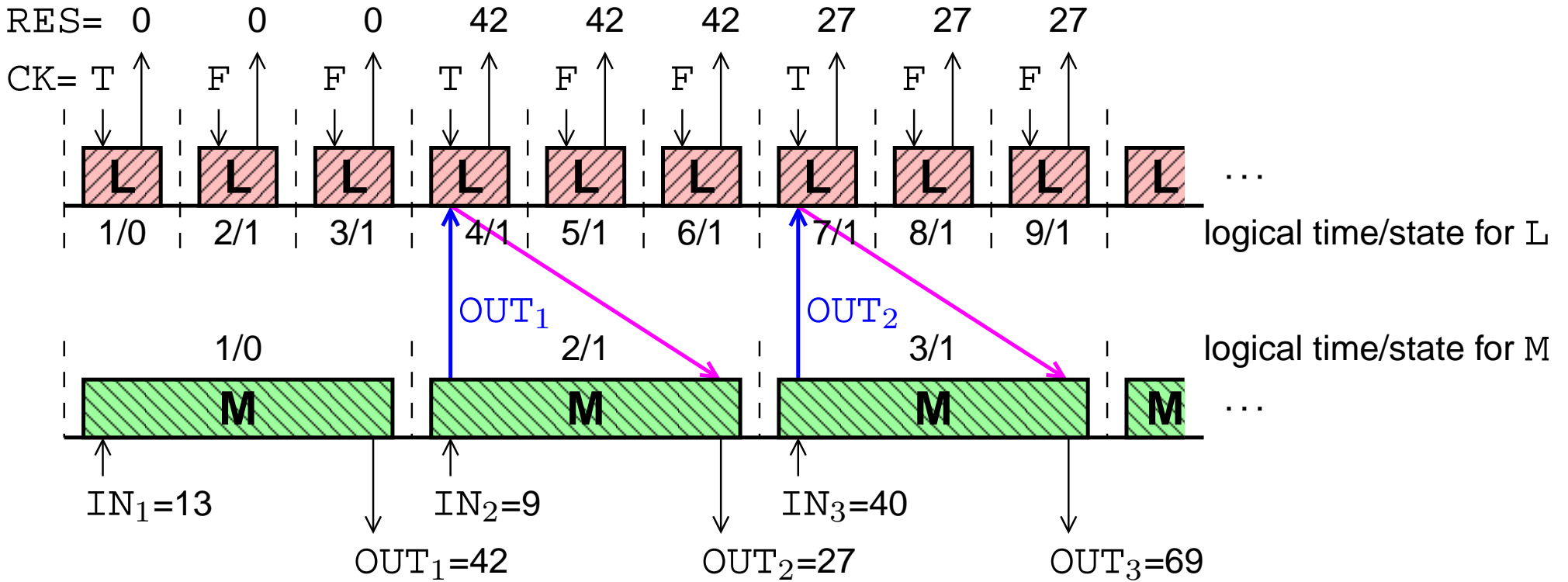
state 1:
go(IN)
send(L,OUT)
OUT:=SLOW(IN)
write(OUT)
goto 1
```

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

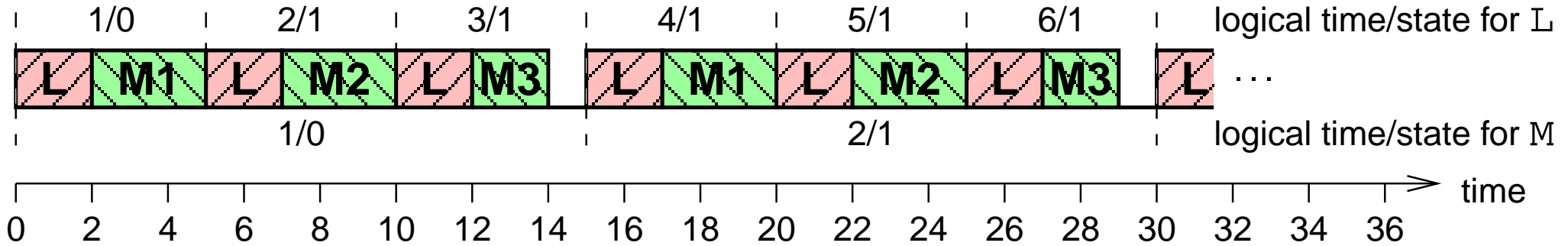
But our program is **distributed** into n tasks. So:

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

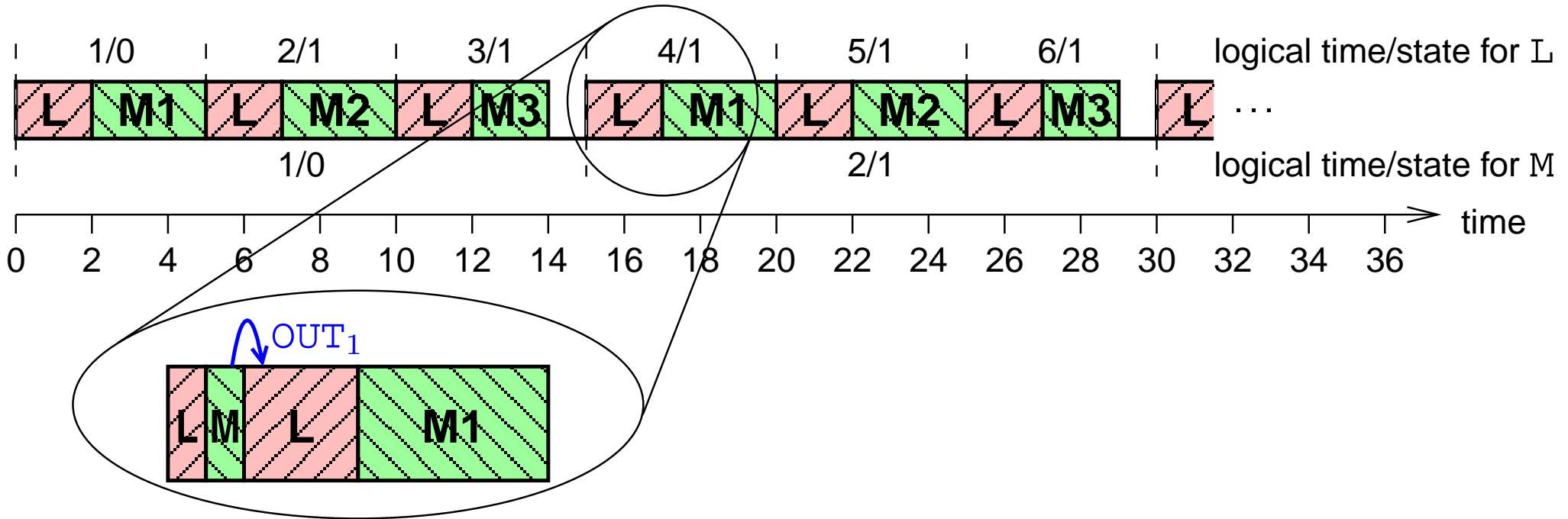
location	L	M
WCET	2	8
rate	5	15

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

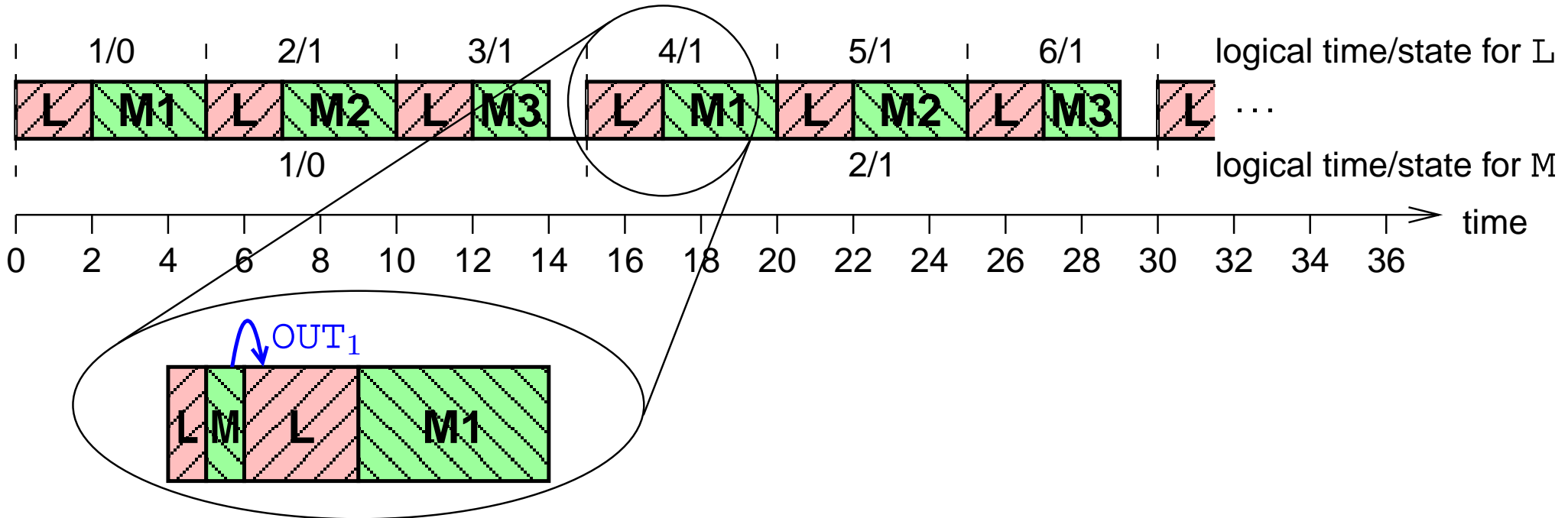
RTOS implementation



RTOS implementation

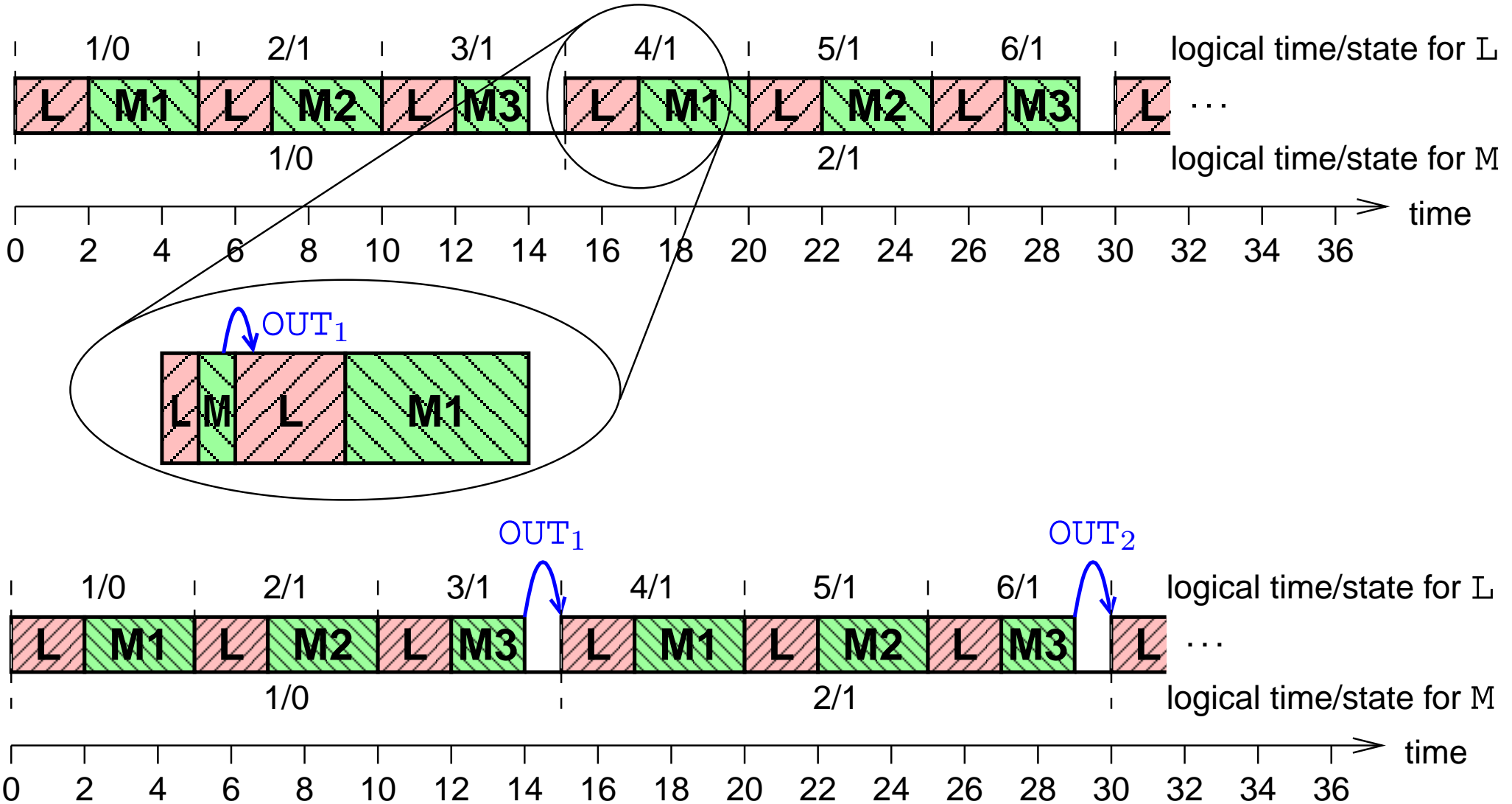


RTOS implementation



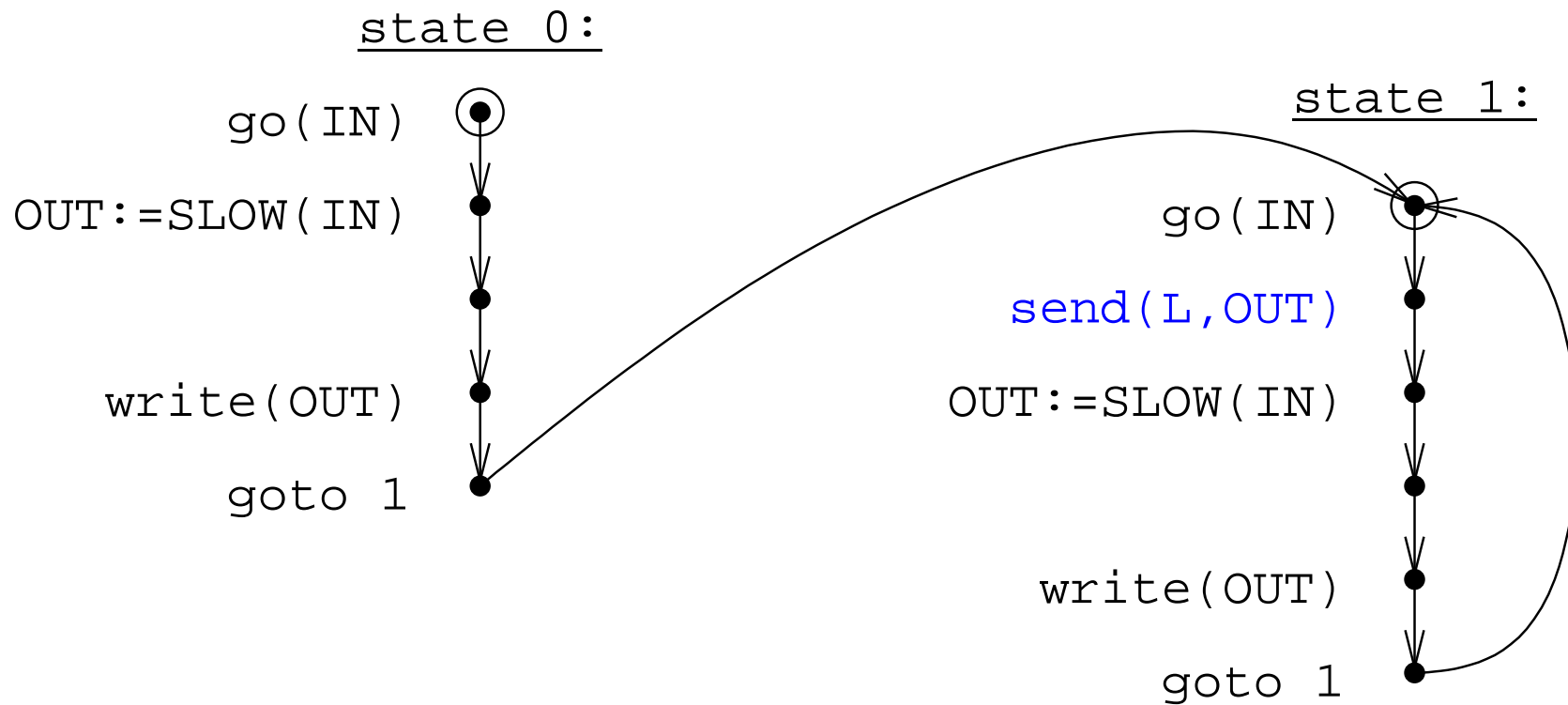
This mechanism relies on the **preemption** mechanism of the RTOS!

RTOS implementation



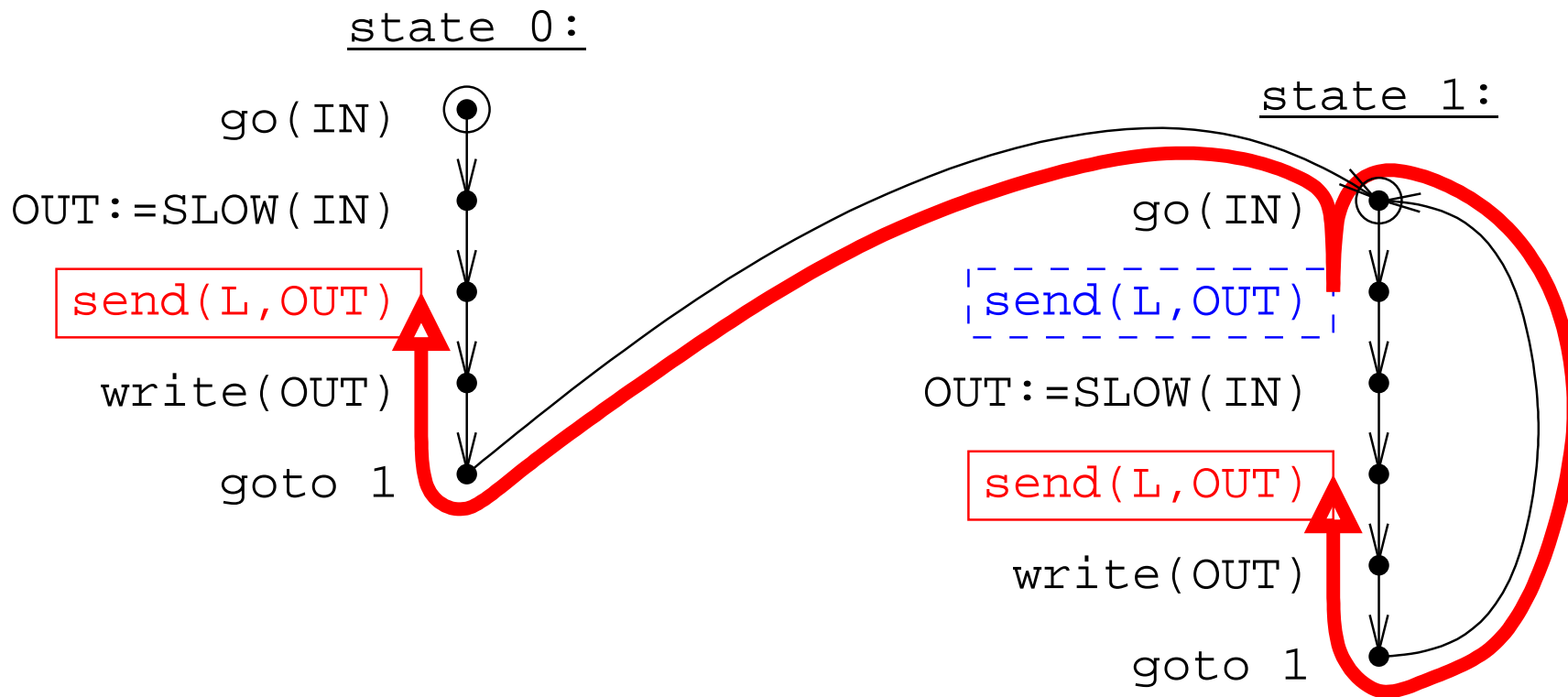
Data-flow analysis

Program of location M



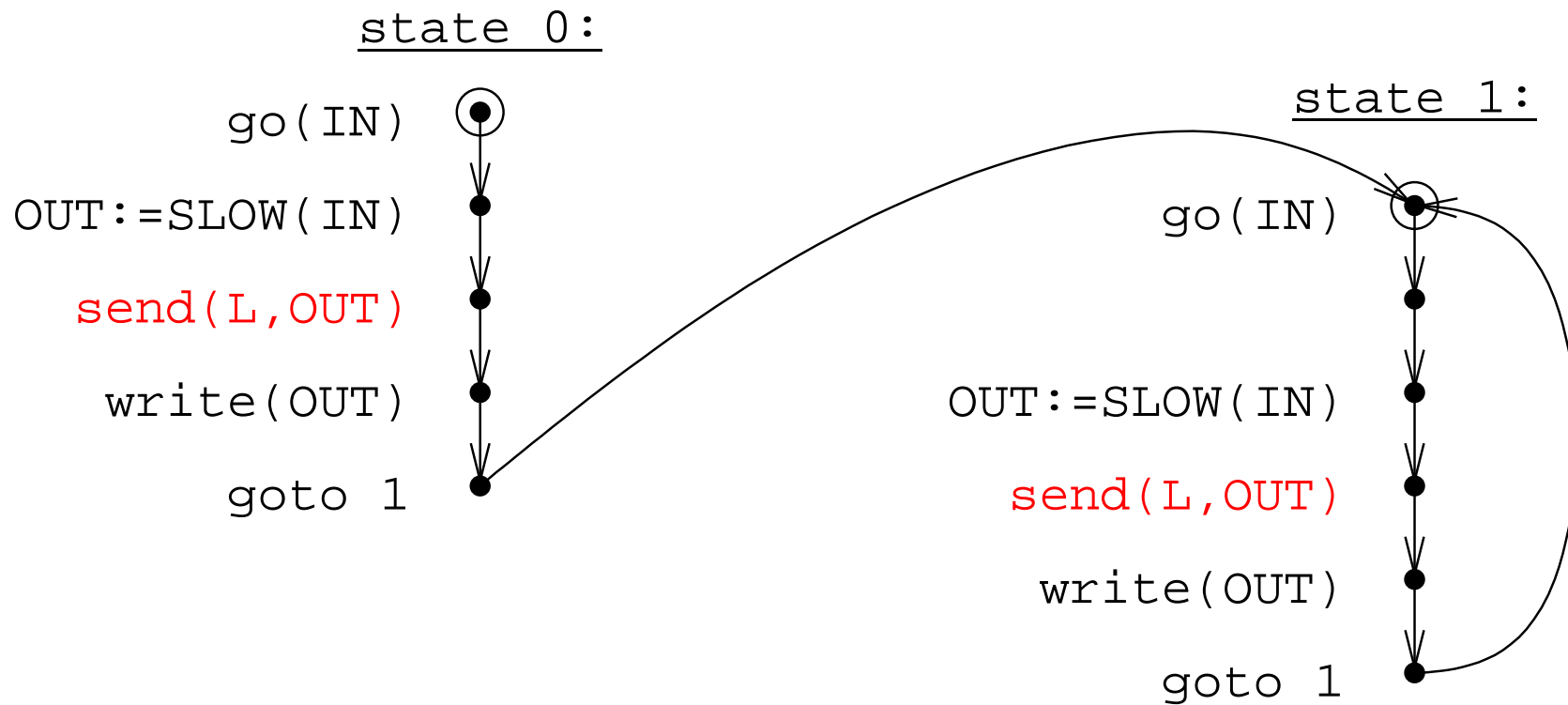
Data-flow analysis

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

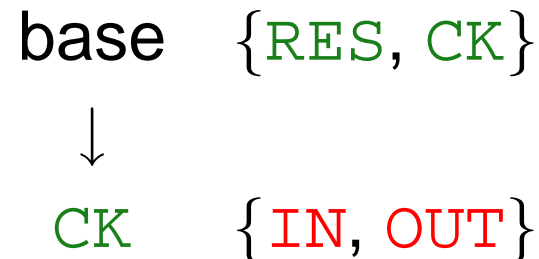
```
node FILTER (CK : bool; (IN : int) when CK)
  returns (RES : int; (OUT : int) when CK);
let
  RES = current ((0 when CK) -> pre OUT);
  OUT = SLOW (IN);
tel.
function SLOW (A : int) returns (B : int);
```

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:



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]

Conclusion and future research

This new distribution method:

- is implemented in the `ocrep` tool:

<http://www.inrialpes.fr/pop-art/people/girault/Ocrep>

- works equally well with `Lustre` and `Esterel` programs
- allows the writing and compiling of synchronous programs with **long duration tasks**

Some future plans:

- To adapt this method to `Decade` programs in order to obtain **code mobility**

`Decade` is a dynamic higher-order synchronous data-flow programming language [Colaço et al 2004]