APPLICATIONS

BRICS
Basic Research
in Computer Science

CENTER FOR INDLEJREDE SOFTWARE SYSTEMER

Overview

- Leader Election Protocol
- Dynamic Voltage Scaling
- Optimal Reconfiguration of FPGA
- Memory Interface

AMETIST
advanced methods for timed systems

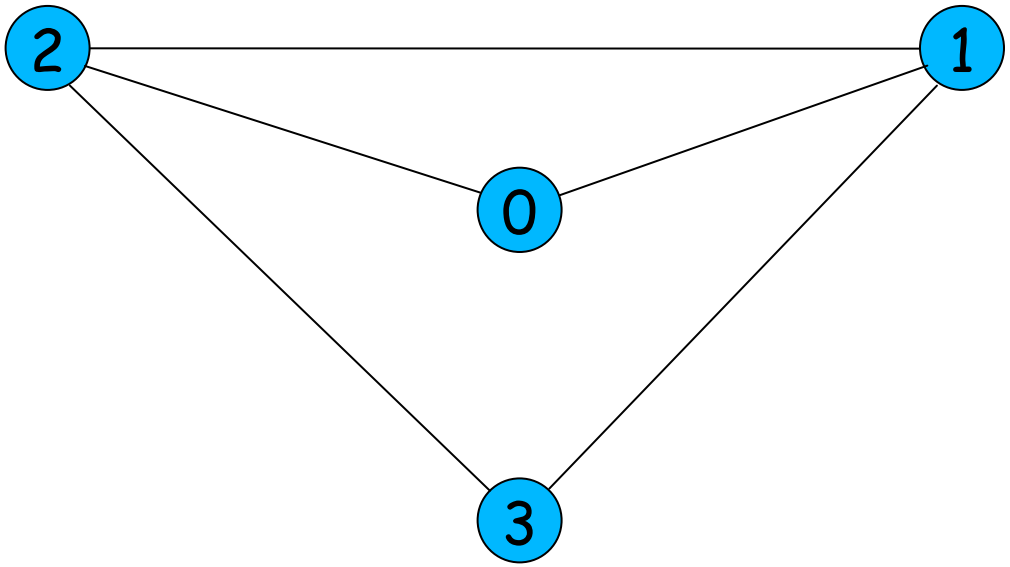
BRICS
Basic Research in Computer Science

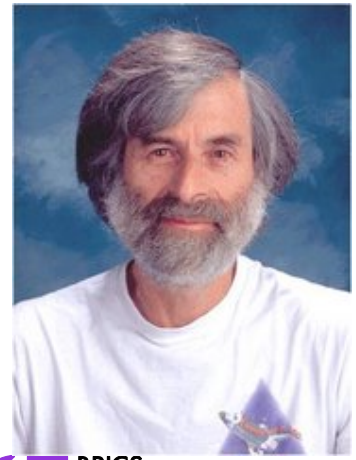
Leader Election Protocol

BRICS
Basic Research
in Computer Science

CSS
CENTER FOR INDLEJREDE SOFTWARE SYSTEMER

Leader Election

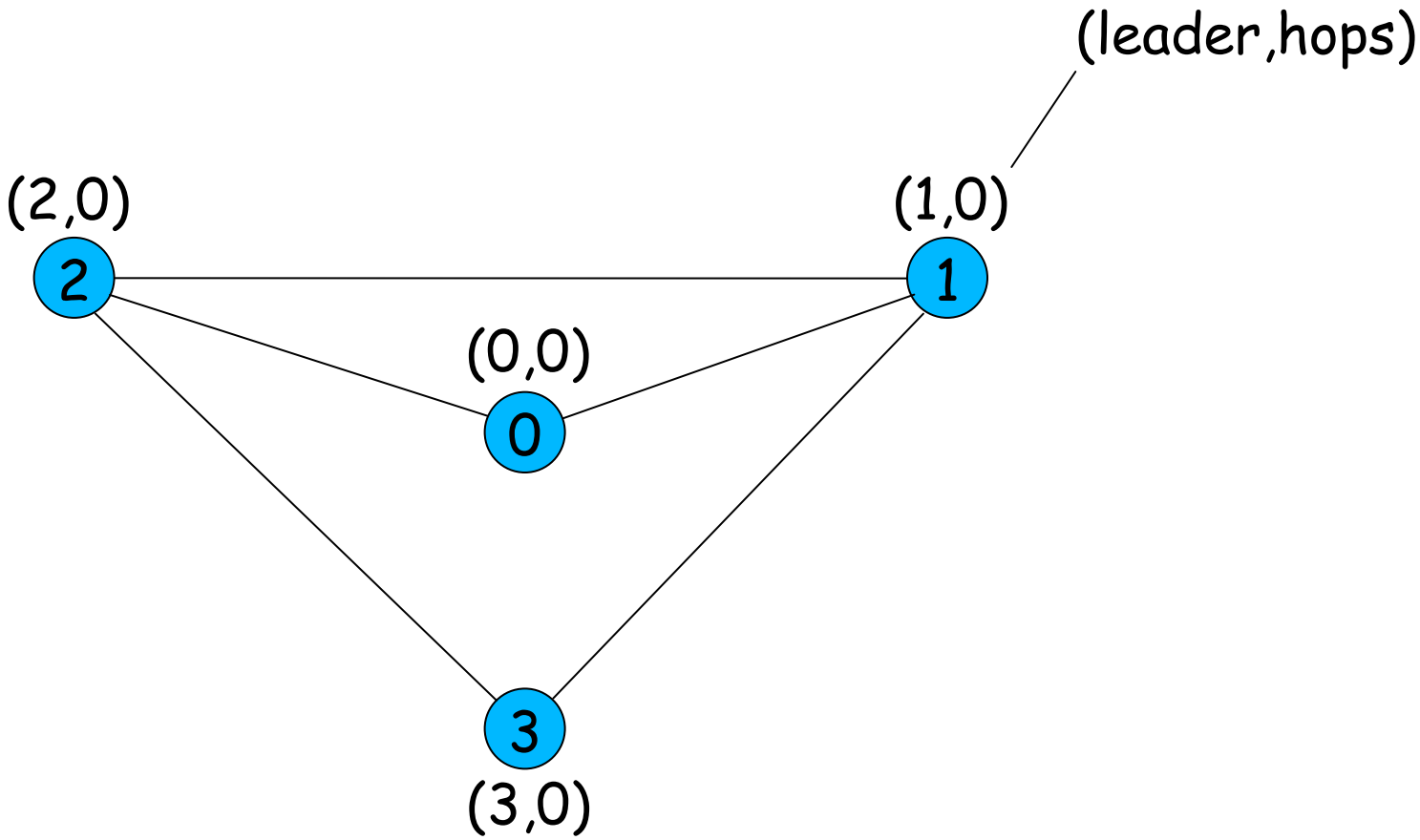




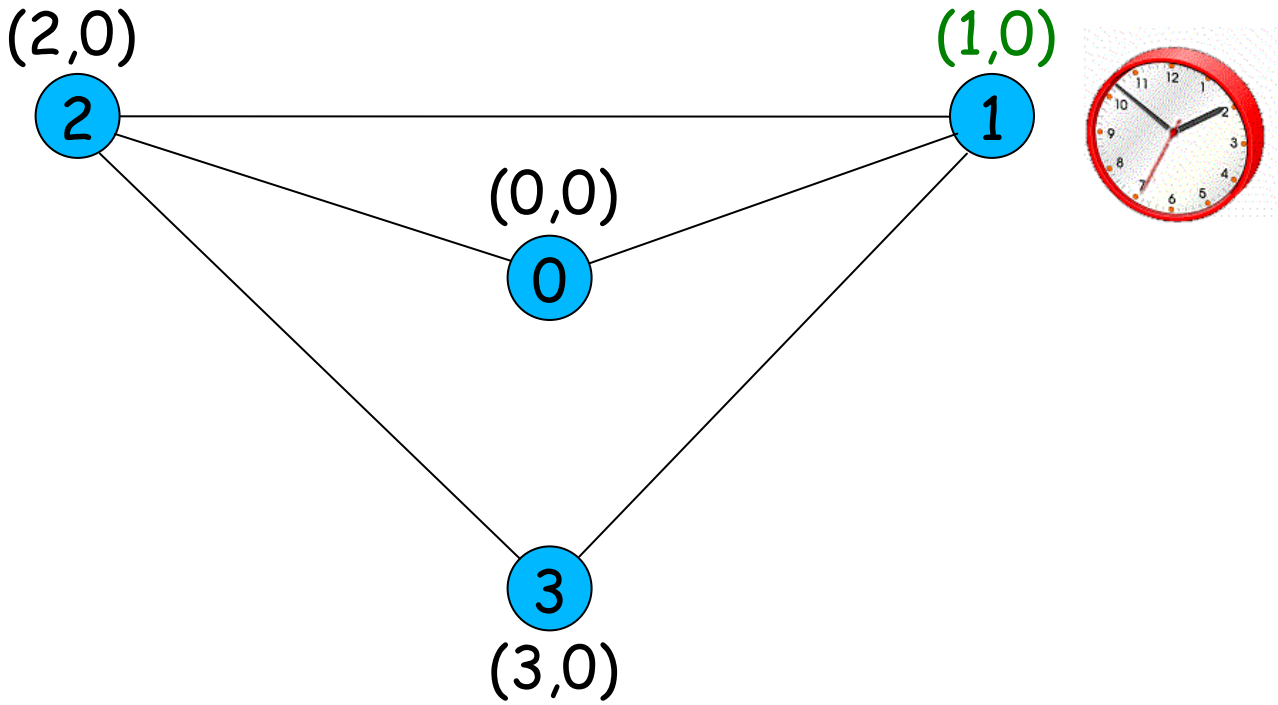
Protocol by
Leslie Lamport

BRICS
Basic Research
in Computer Science

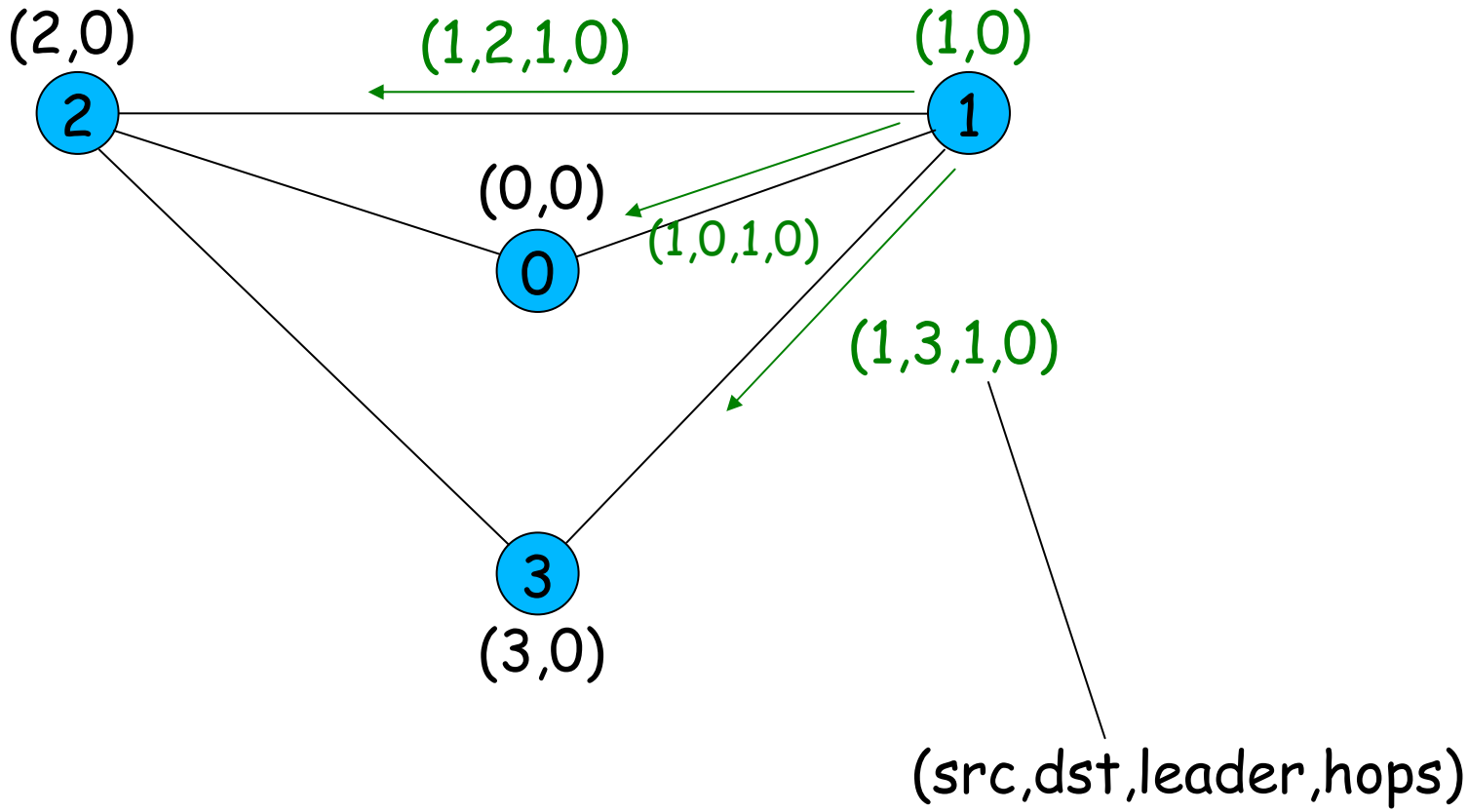
Leader Election



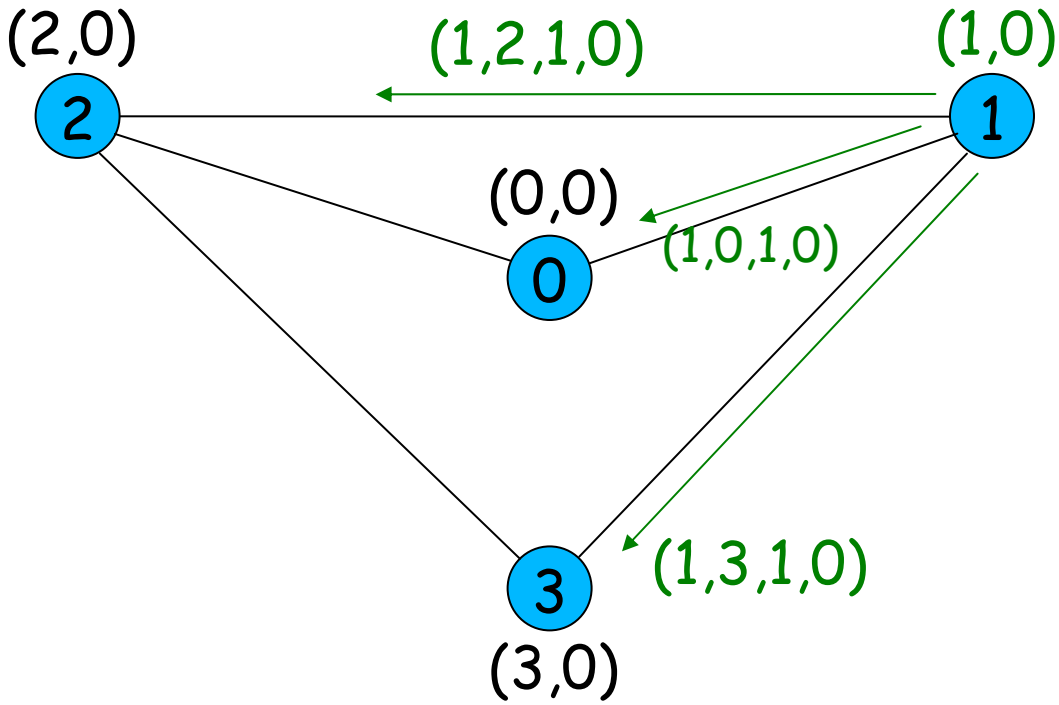
Timeout



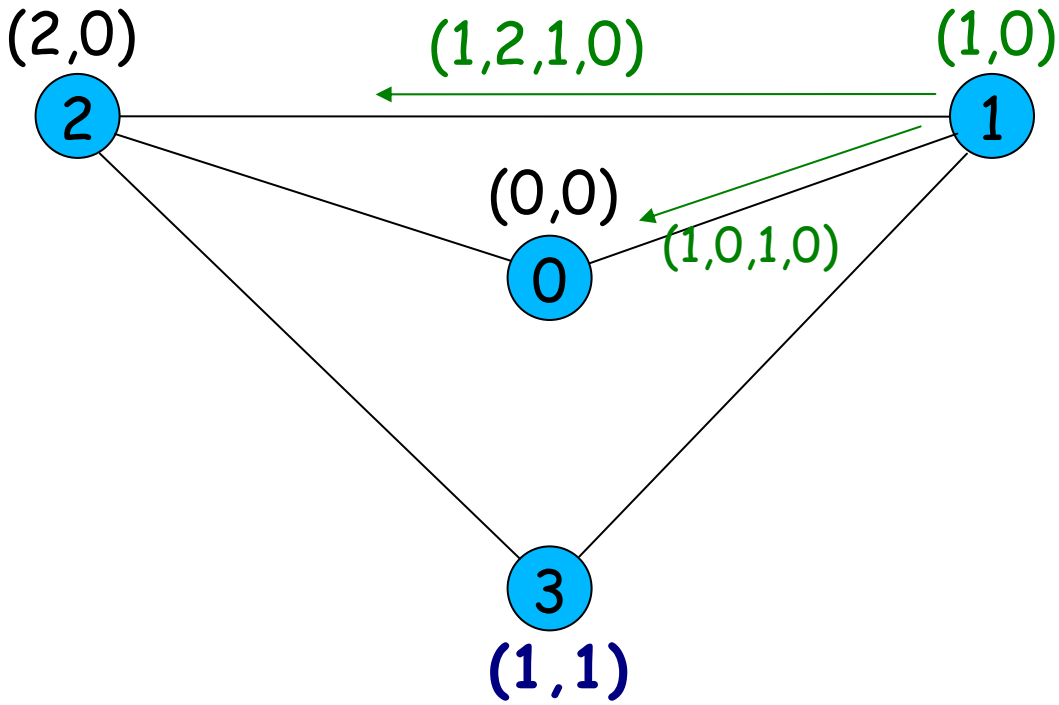
Flooding



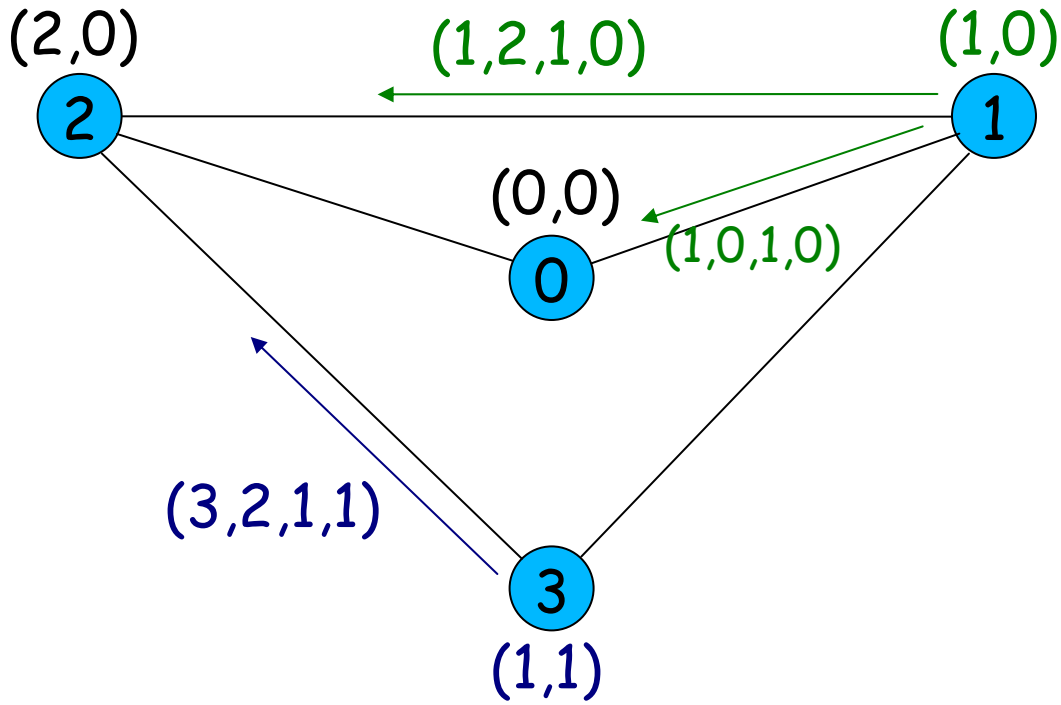
Flooding



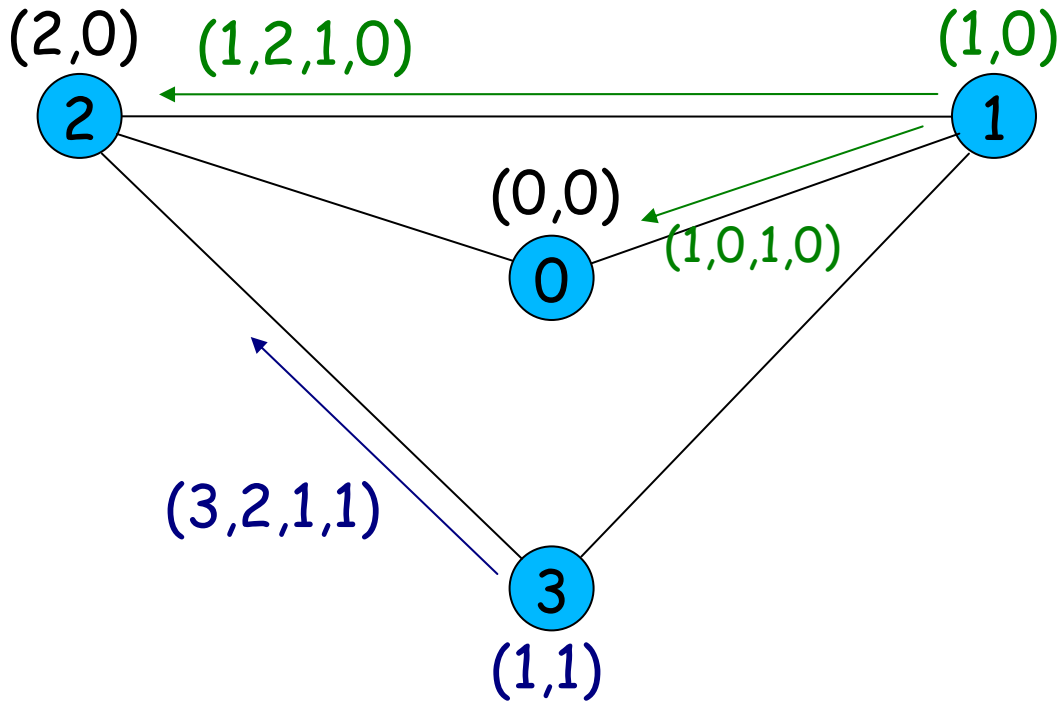
Flooding



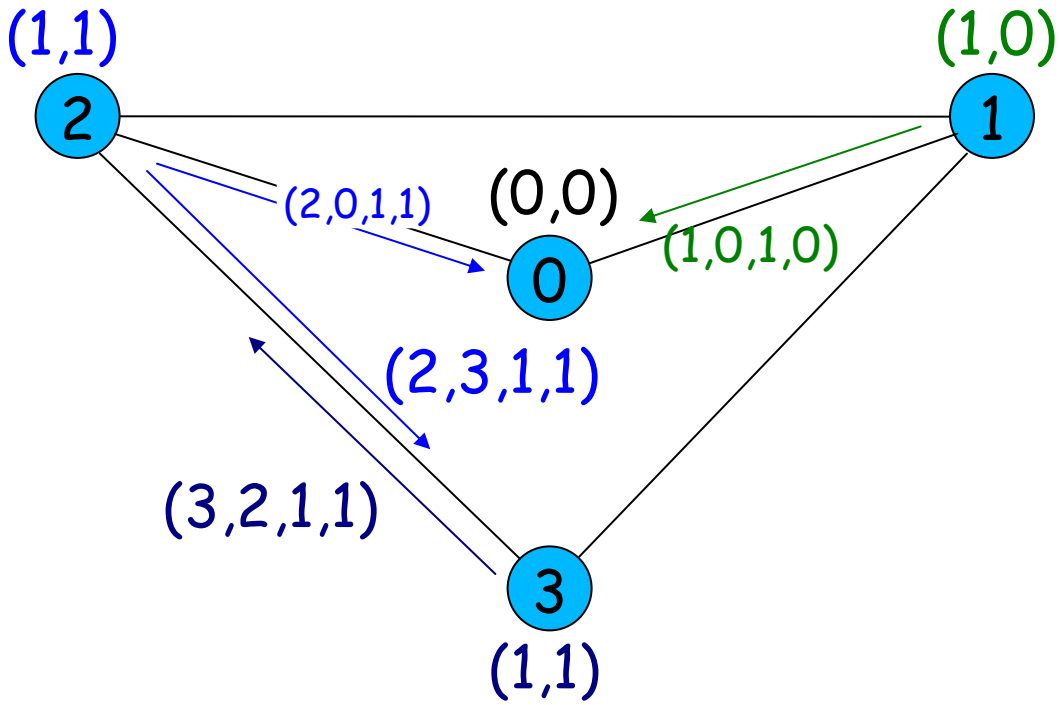
Forwarding



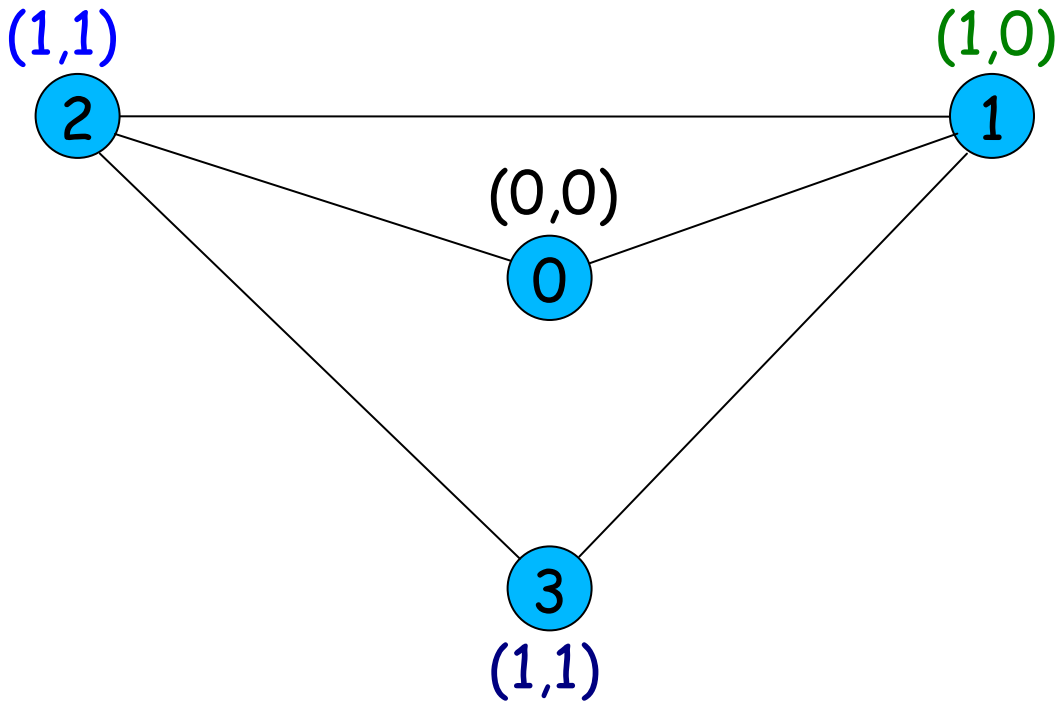
Forwarding



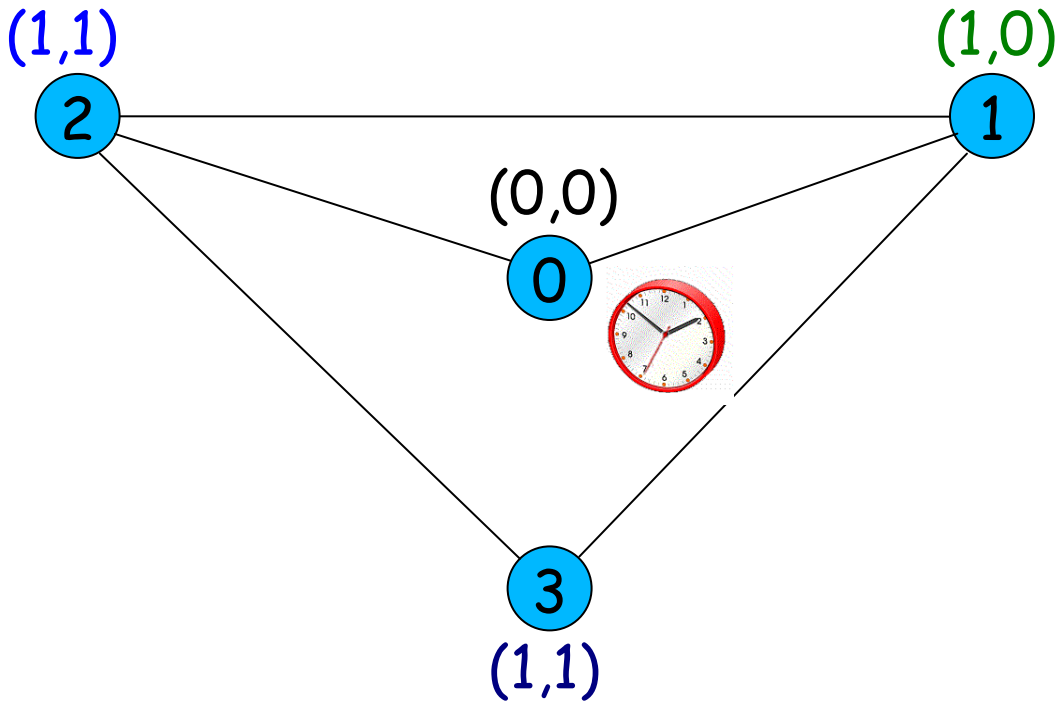
Forwarding



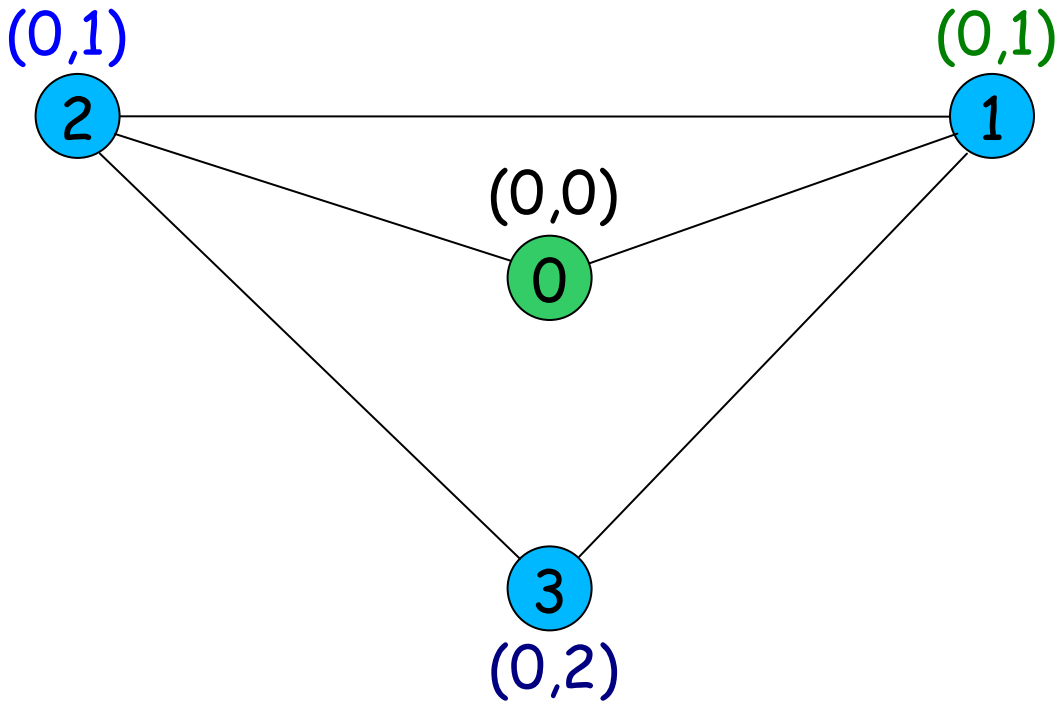
Leader Election



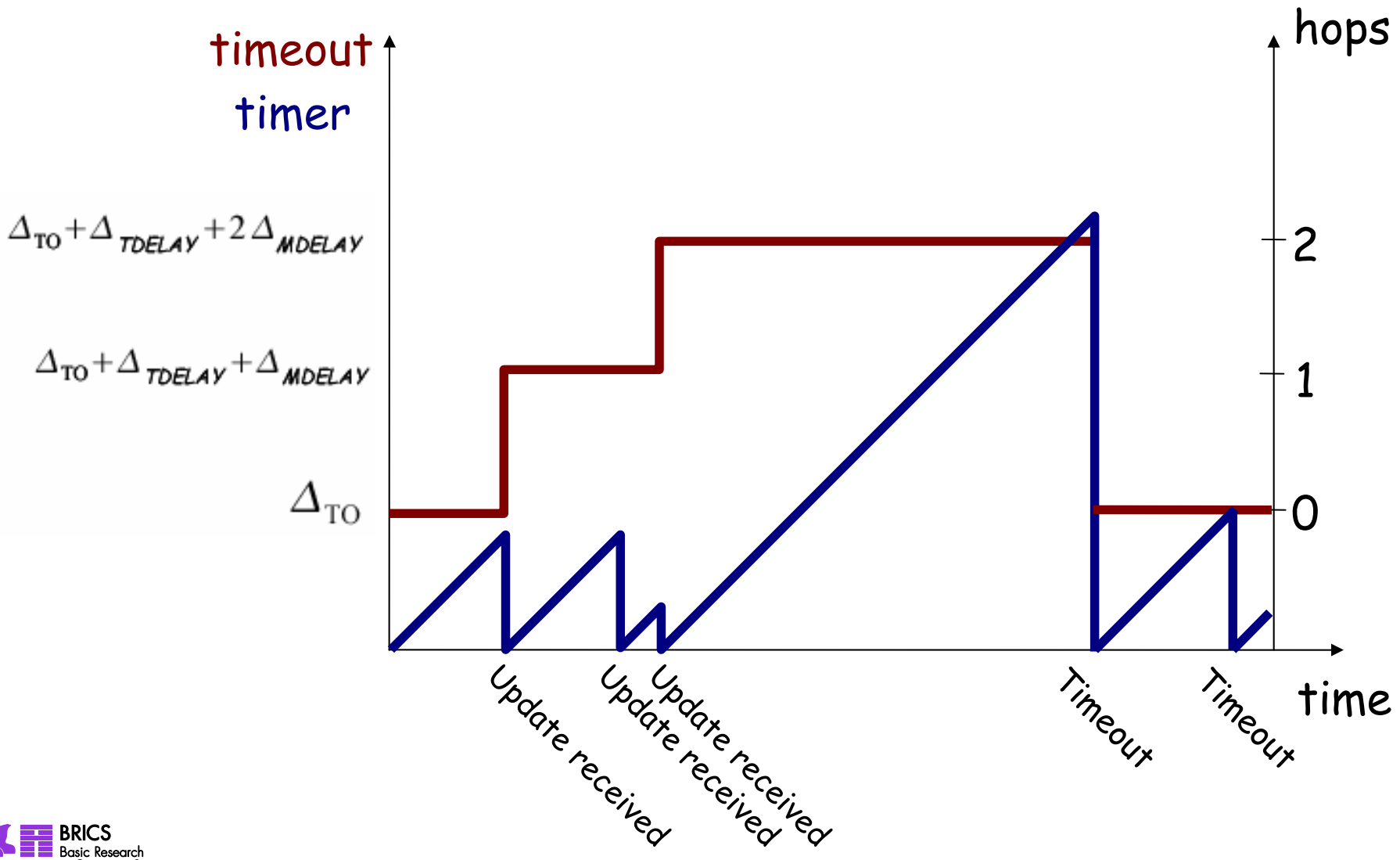
Leader Election



Leader Election



Variable timeout



Leader Election

Claim to be verified

Correct leader is known at a node *i* after

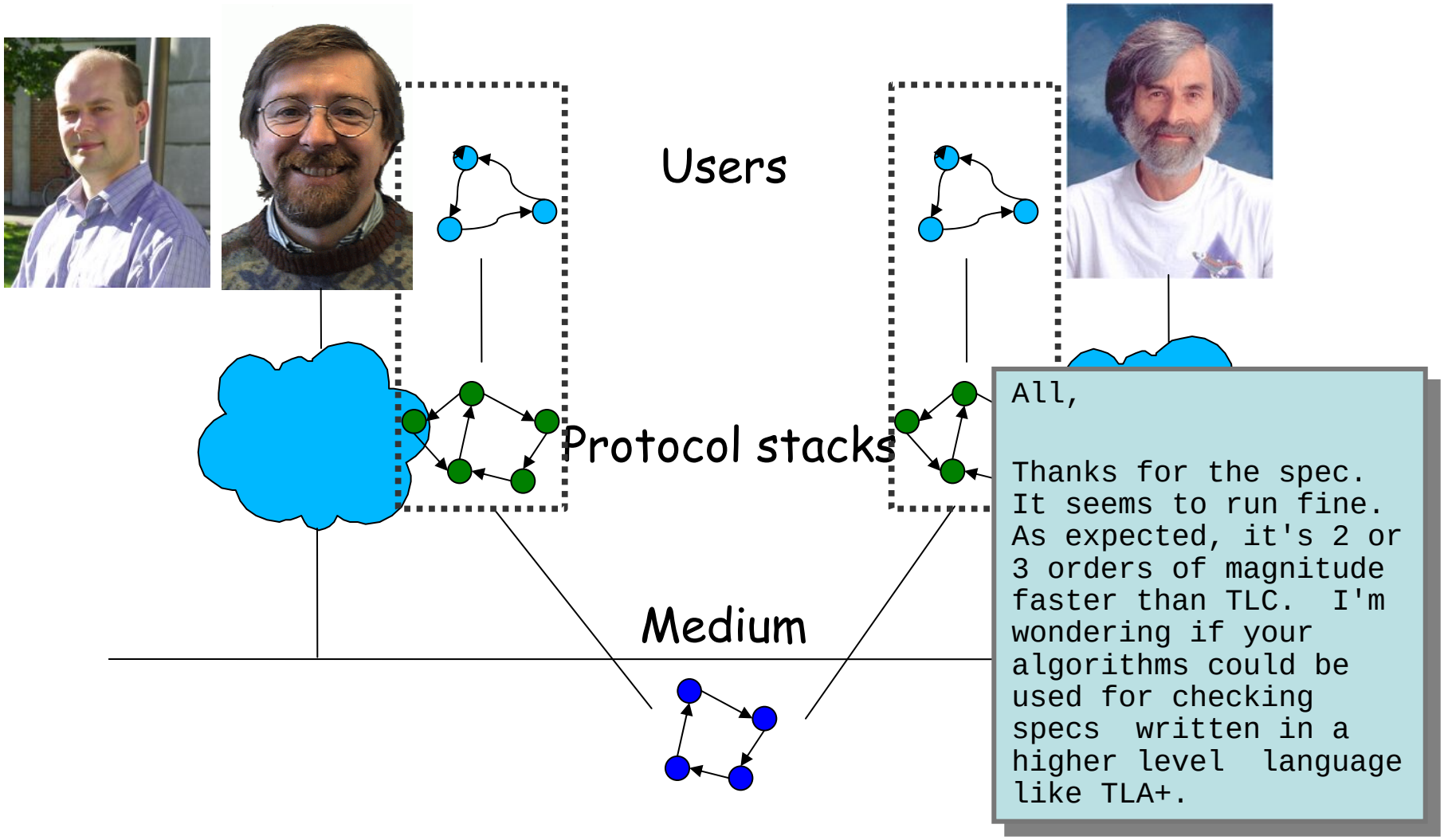
$$t(i) = \Delta_{TO} + \Delta_{TDELAY} + d_i \cdot \Delta_{MDELAY}$$

A model checking problem

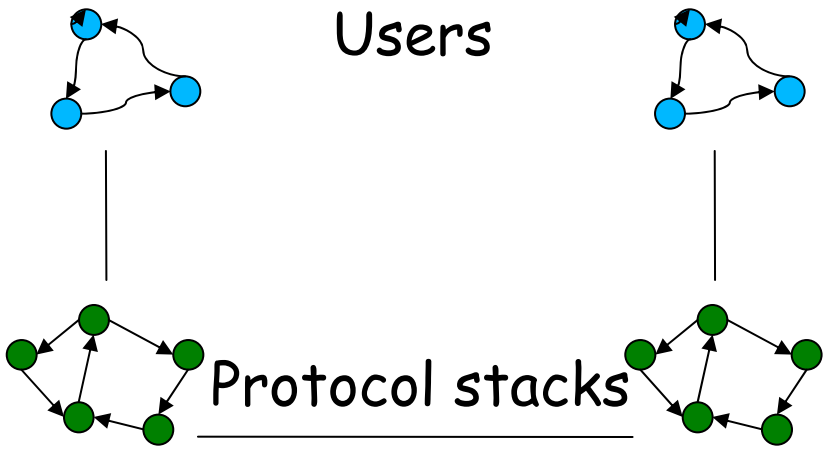
$$IMP \models \Box_{>t(i)}\ l(i)=L(i)$$

for all i.

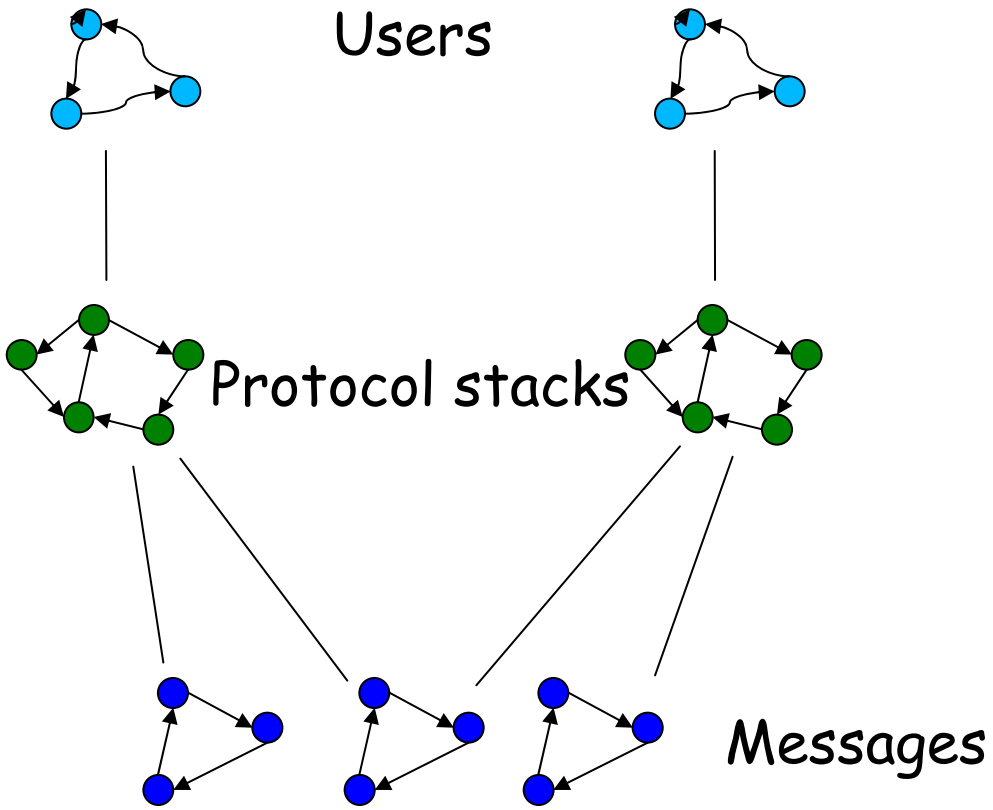
Modelling (RT) protocols



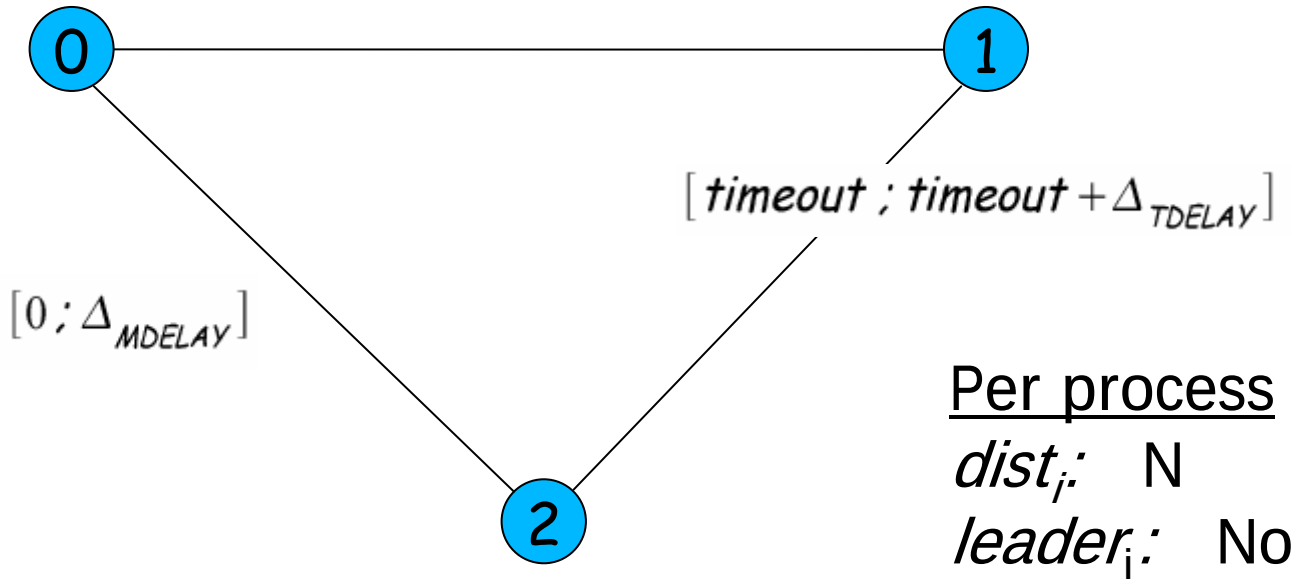
Modelling (RT) protocols



Modelling (RT) protocols



Modelling the election protocol



Per process

$dist_i$: N

$leader_i$: Node

$timeout_i$: N

Static

$Topology : Node \times Node \rightarrow B$

Message

src: Node

dst: Node

leader: Node

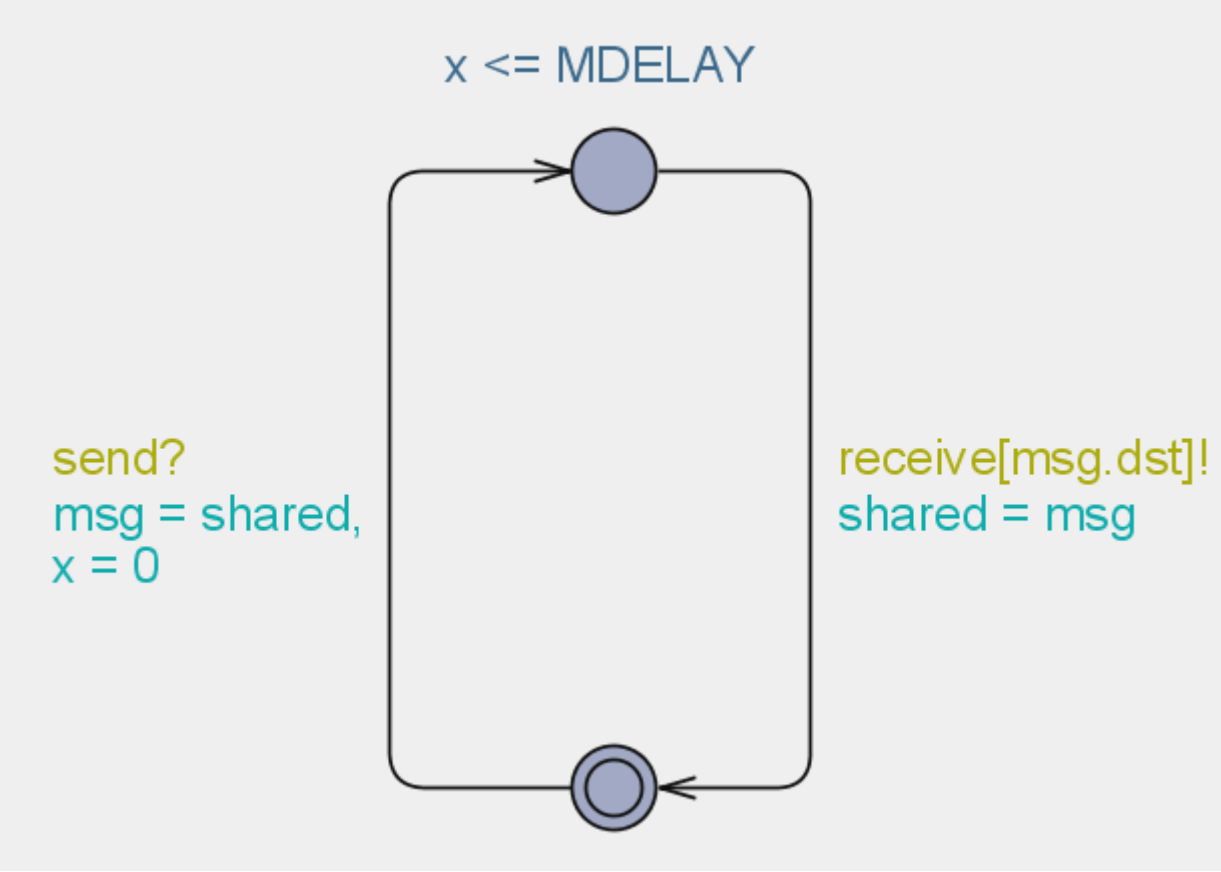
hopss: N

Global Declaration

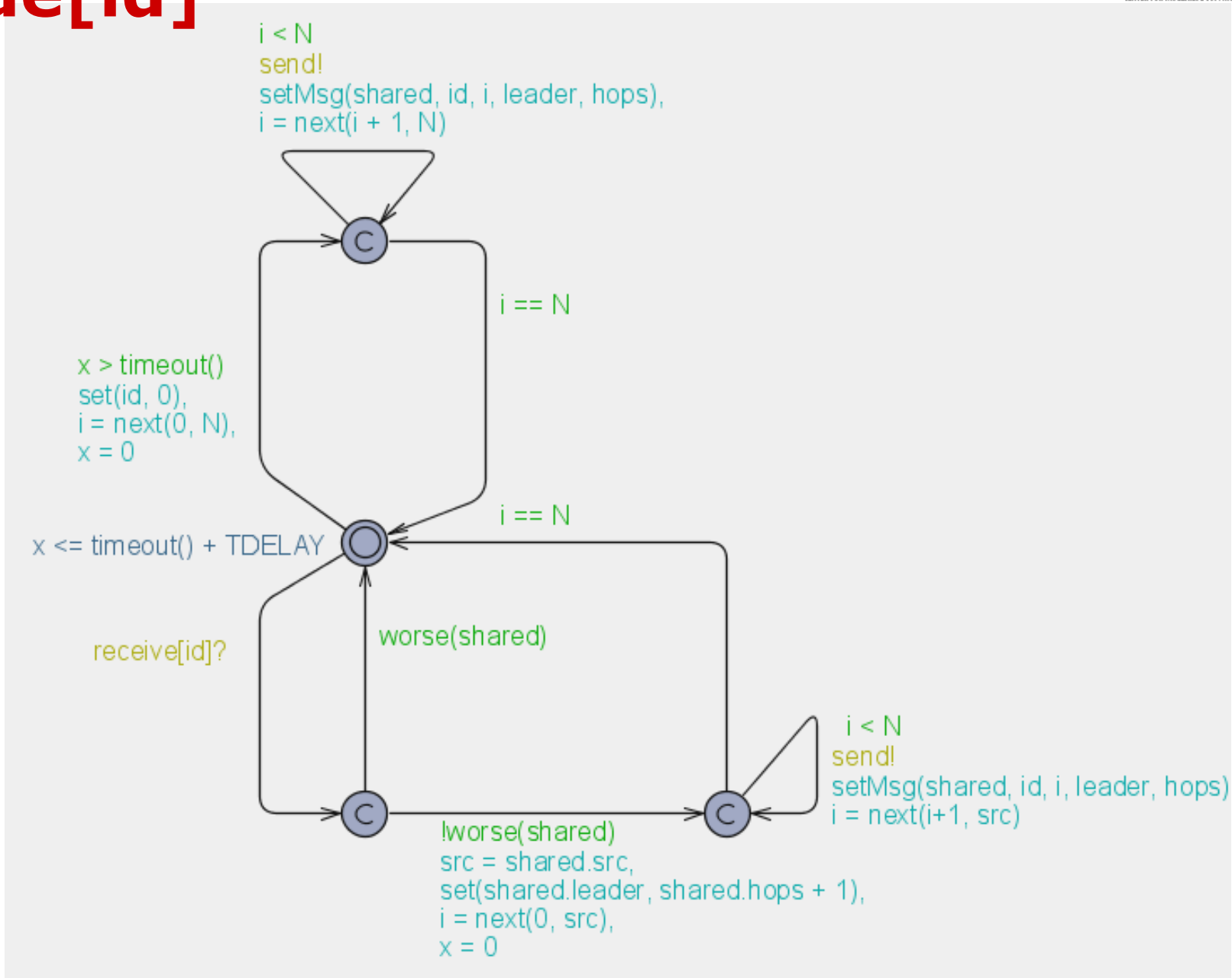
```
void setMsg(msg_t &msg, id_t src, id_t dst, id_t leader, int[0,N] hops)
{
  msg.src = src;
  msg.dst = dst;
  msg.leader = leader;
  msg.hops = hops;
}

chan send;
chan receive[N];
msg_t shared;

const int link[N][N] = {
  { 0,1,1 },
  { 1,0,1 },
  { 1,1,0 }
};
```

```
const int N = 3;
const int MDELAY = 3;
const int TDELAY = 5;
const int TO = 10;

typedef int[0,N-1] id_t;
typedef struct
{
  id_t src;
  id_t dst;
  id_t leader;
  int[0,N] hops;
} msg_t;
```

Message



Node[id]


i < N
send!
setMsg(shared, id, i, leader, hops),
i = next(i + 1, N)
i == N
x > timeout()
set(id, 0),
i = next(0, N),
x = 0
i == N
x <= timeout() + TDELAY
receive[id]?
worse(shared)
i < N
send!
setMsg(shared, id, i, leader, hops)
i = next(i+1, src)
!worse(shared)
src = shared.src,
set(shared.leader, shared.hops + 1),
i = next(0, src),
x = 0

Local Declarations (Node[id])

```
id_t leader = id;
int[0,N] hops;
clock x;
int[0,N] i;
id_t src;

void set(id_t l, int[0,N] h)
{
  leader = l;
  hops = h;
}

int[0,N] next(int[0,N] i,int[0,N] src)
{
  while (i  < N && (!link[id][i] || i == src))
  {
    i++;
  }
  return i;
}
```

```
int[0,1000] timeout()
{
  if (hops > 0)
    return TO + TDELAY + hops * MDELAY;
  return TO;
}

bool worse(const msg_t &msg)
```

Optimisations

- Reducing the number of active variables
  - If variable is never used until next reset, then the value does not matter.

- Symmetry of message processes
  - The message processes are symmetric: It does not matter which is used to transfer a message.

Dynamic Voltage Scaling

BRICS
Basic Research
in Computer Science

CISS
CENTER FOR INDLEJREDE SOFTWARE SYSTEMER

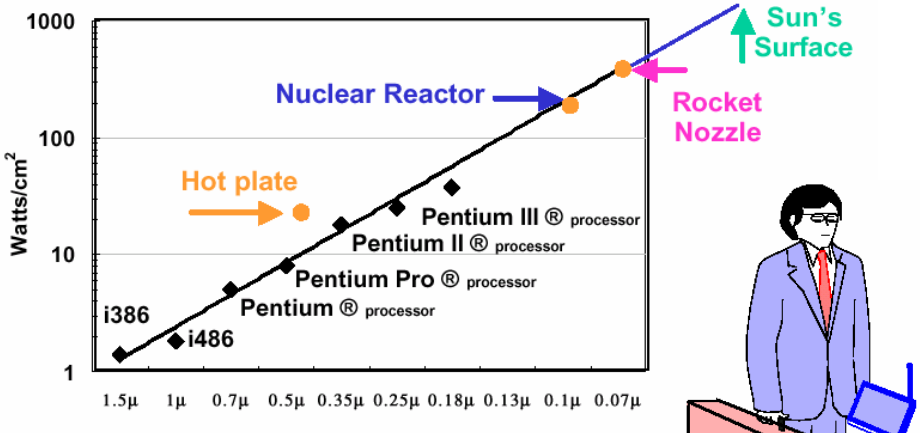
Performance vs Ressource-Efficiency ?

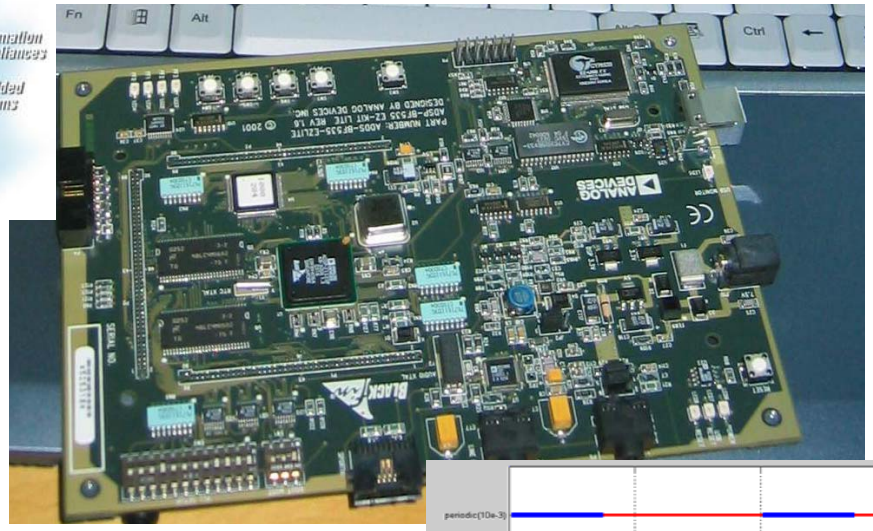
- Consumer constantly demand
  better functionality, flexibility, availability, …
- .. increase in resources needed:
  - Time
  - Energy
  - Memory
  - Bandwidth
  - ..
- Application of **CUPPAAL** to modeling, analysis and synthesis of resource-efficient schedules for real-time systems.

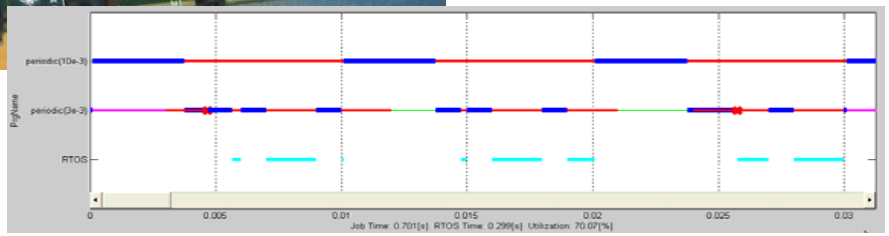
BRICS
Basic Research
in Computer Science

Power Management

Dynamic Voltage Scaling





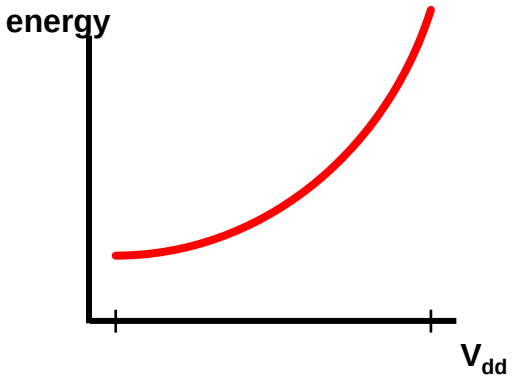


BRICS
Basic Research
in Computer Science

Energy in Processor

A non-experts understanding of CMOS

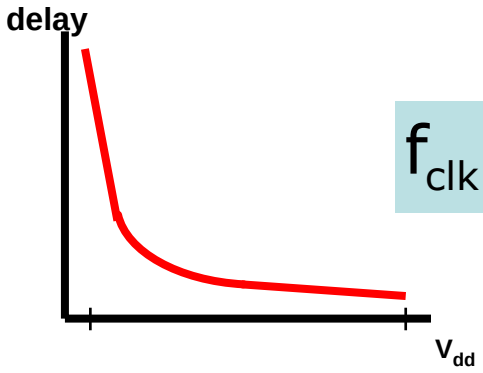
- Power consumption mainly by dynamic power



$$P_{dynamic} = C_L \cdot V_{dd}^2 \cdot f_{clk}$$

$$E_{dynamic\ pr.\ cycle} = C_L \cdot V_{dd}^2$$

- Supply voltage reduction => decreased frequency



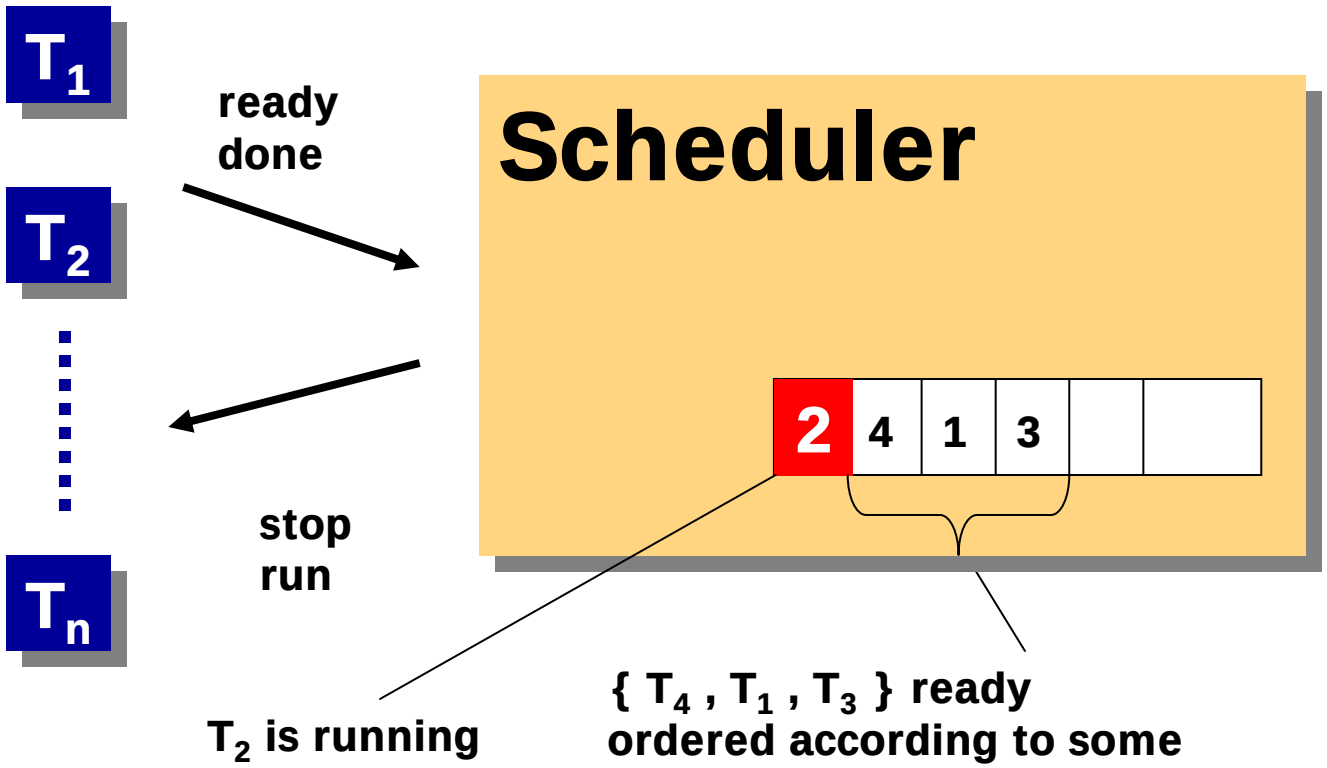
$$f_{clk} \sim V_{dd}^{(\alpha\text{-}1)} \quad ;\alpha > 1$$

We may miss deadlines

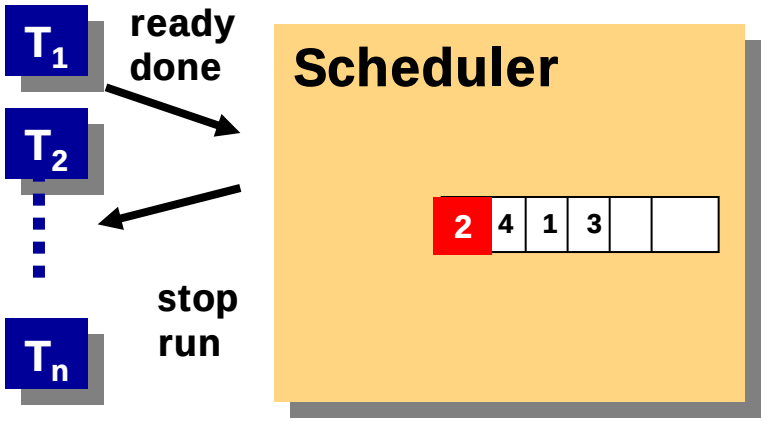
Task Scheduling

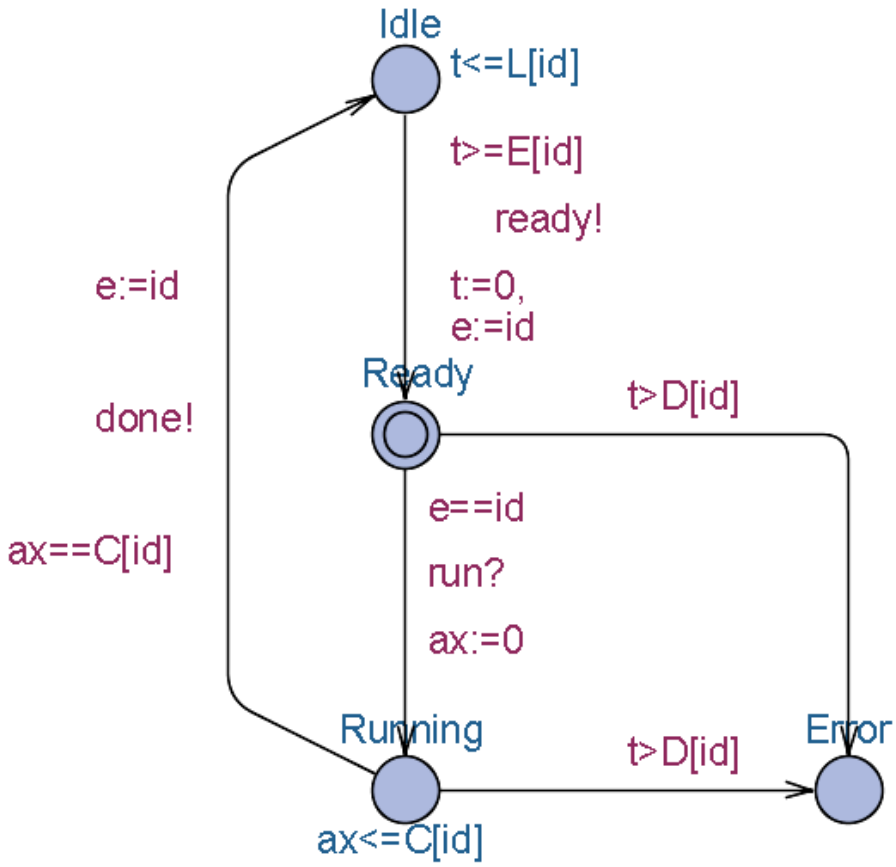
utilization of CPU

P(i), [E(i), L(i)], .. : period or
earliest/latest arrival or .. for T_i
C(i): execution time for T_i
D(i): deadline for T_i

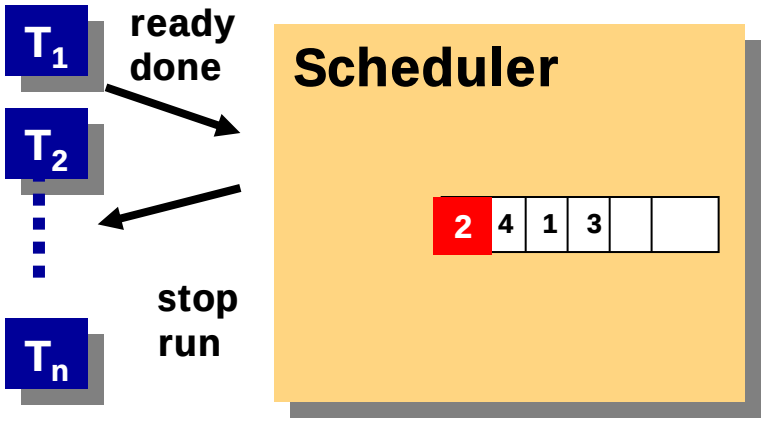


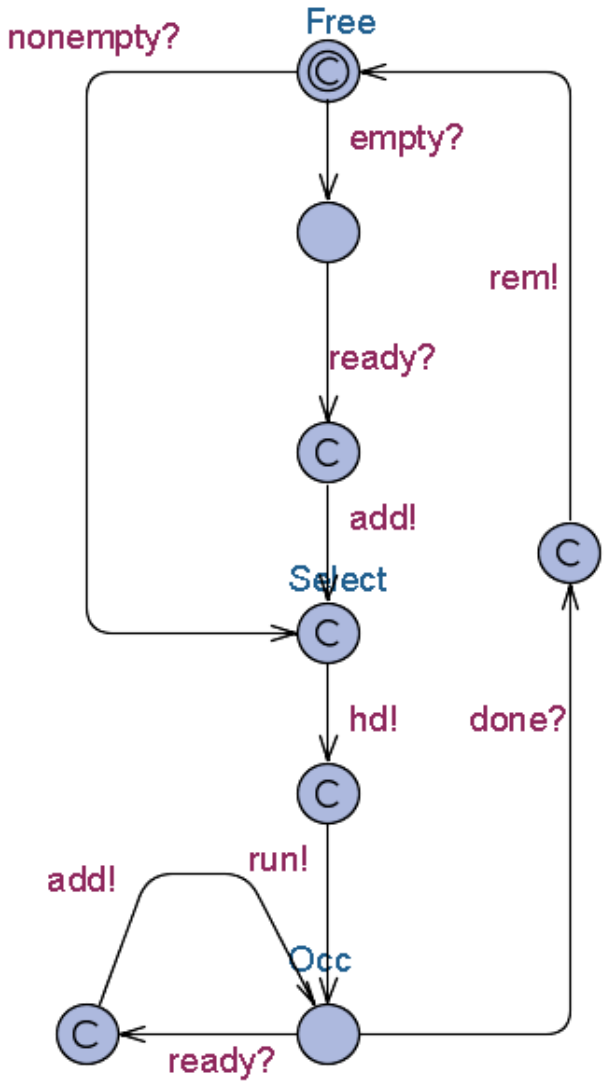
Modeling Task



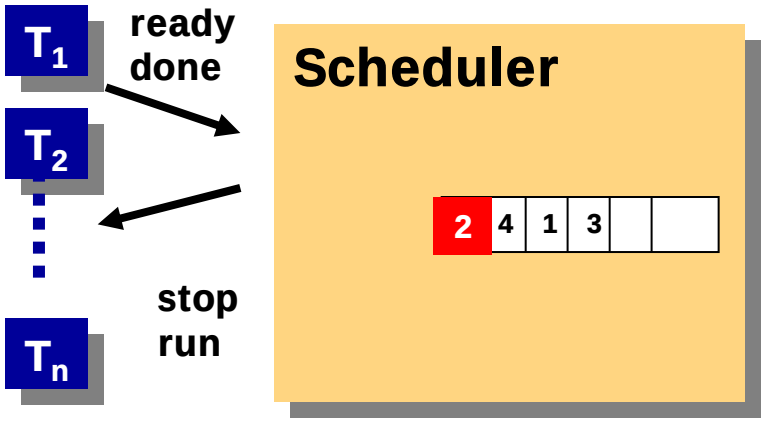


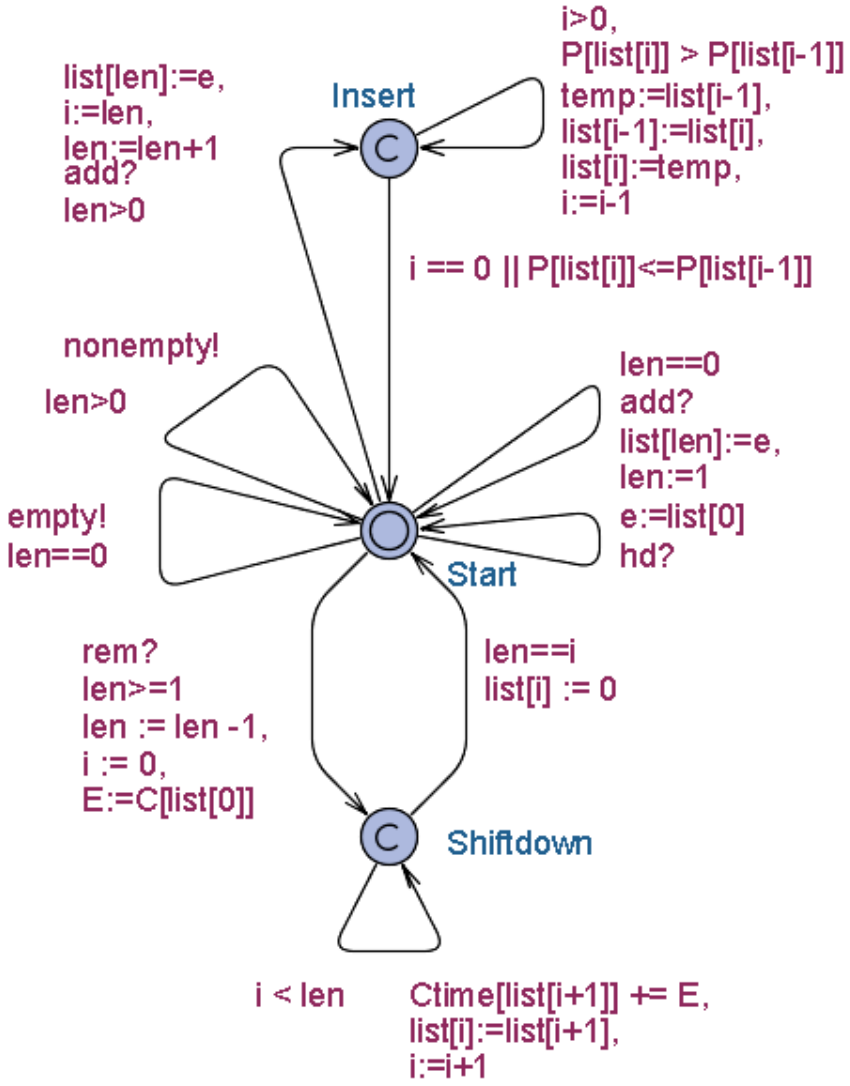
Modeling Scheduler





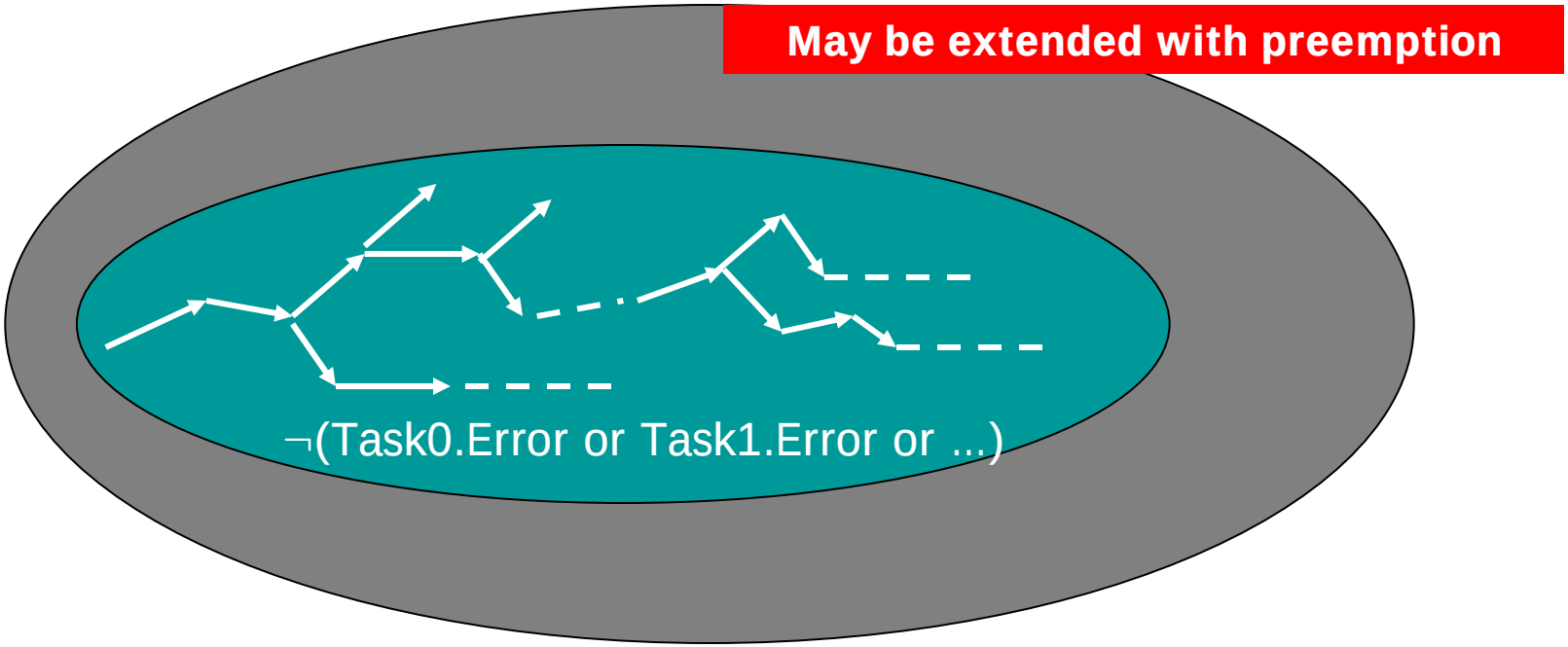
Modeling Queue





BRICS
Basic Research
in Computer Science

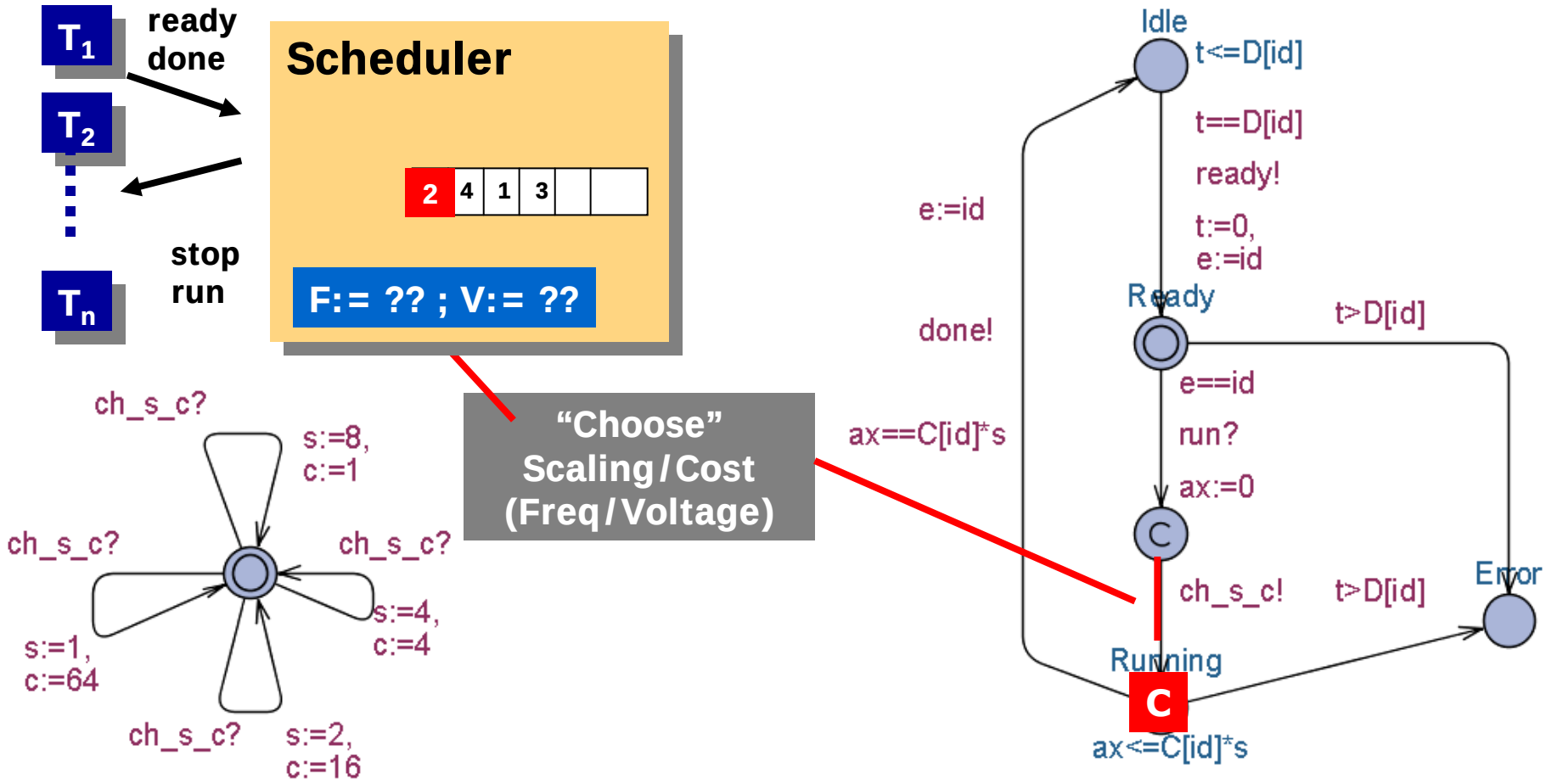
Schedulability = Safety Property



A□ ¬(Task0.Error or Task1.Error or ...)

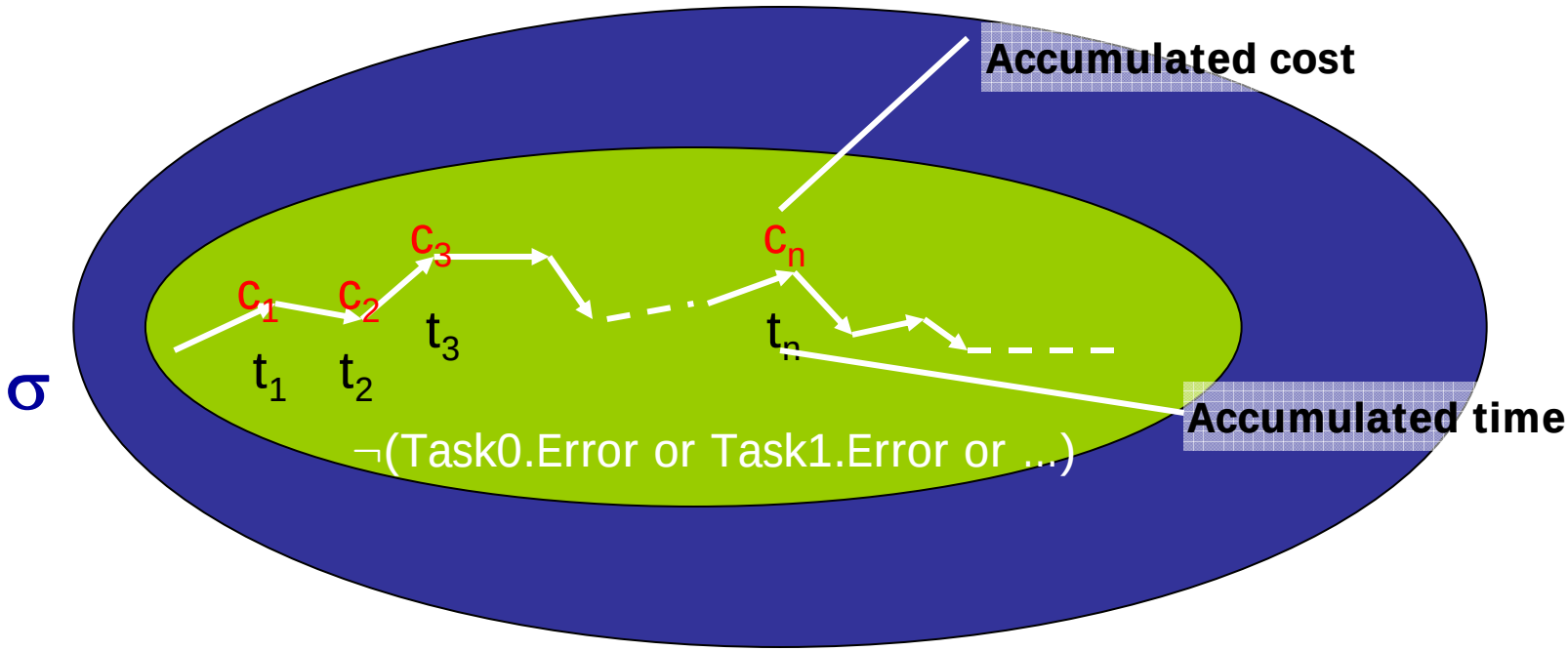
Energy Optimal Scheduling

Using Priced Timed Automata



Energy Optimal Scheduling =

Optimal Infinite Path

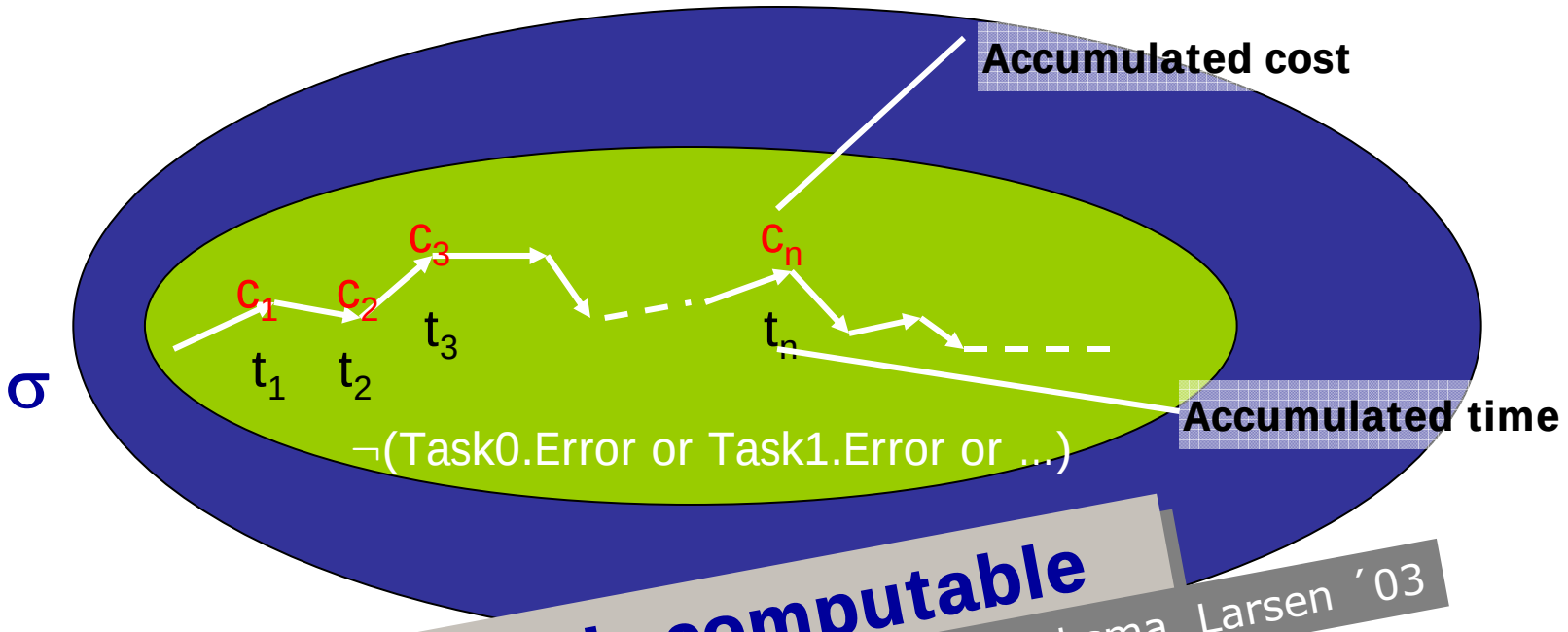


Value of path σ: $val(\sigma) = \lim_{n\to\infty} c_n/t_n$

Optimal Schedule σ*: $val(\sigma^*) = \inf_\sigma val(\sigma)$

Energy Optimal Scheduling =

Optimal Infinite Path



THEOREM: σ^* is computable

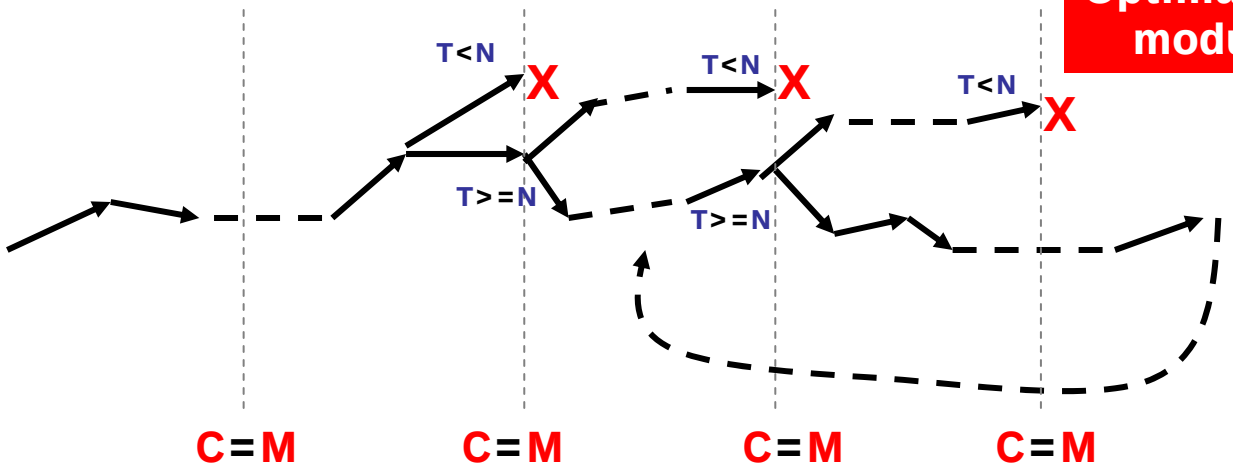
Bouyer, Brinksma, Larsen '03

path σ: $val(\sigma) = \lim_{n\to\infty} c_n/t_n$

Optimal Schedule σ^*: $val(\sigma^*) = \inf_\sigma val(\sigma)$

Approximate Optimal Schedule

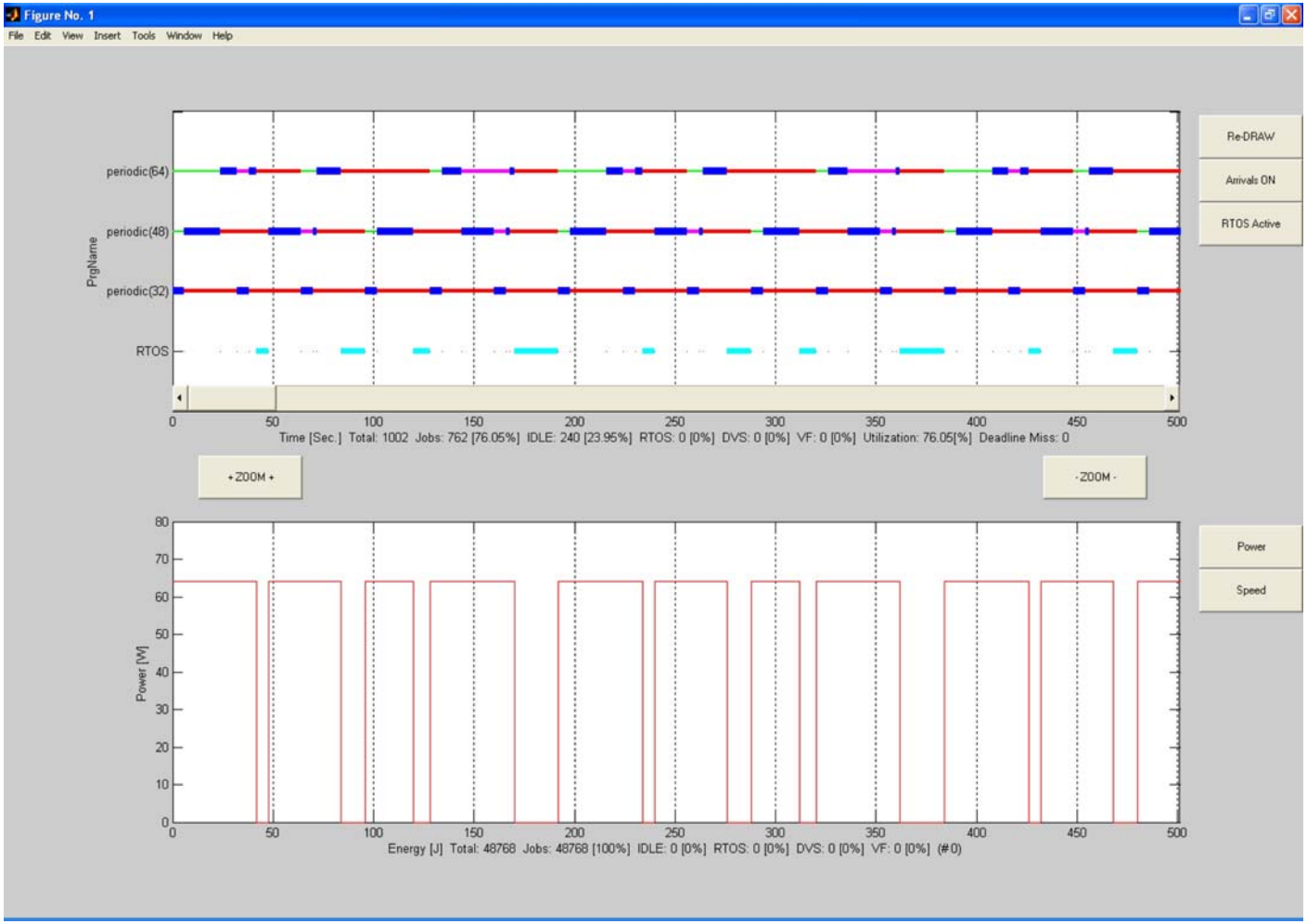
Optimal infinite schedule modulo cost-horizon



E[] (not (Task0.Error or Task1.Error or Task2.Error)
and
(cost>=M imply time >= N))
=
E[] ϕ(M ,N)

$\sigma \models [] \phi(M,N)$ **imply val(σ)$\leq$ M/N**

Preliminary Results

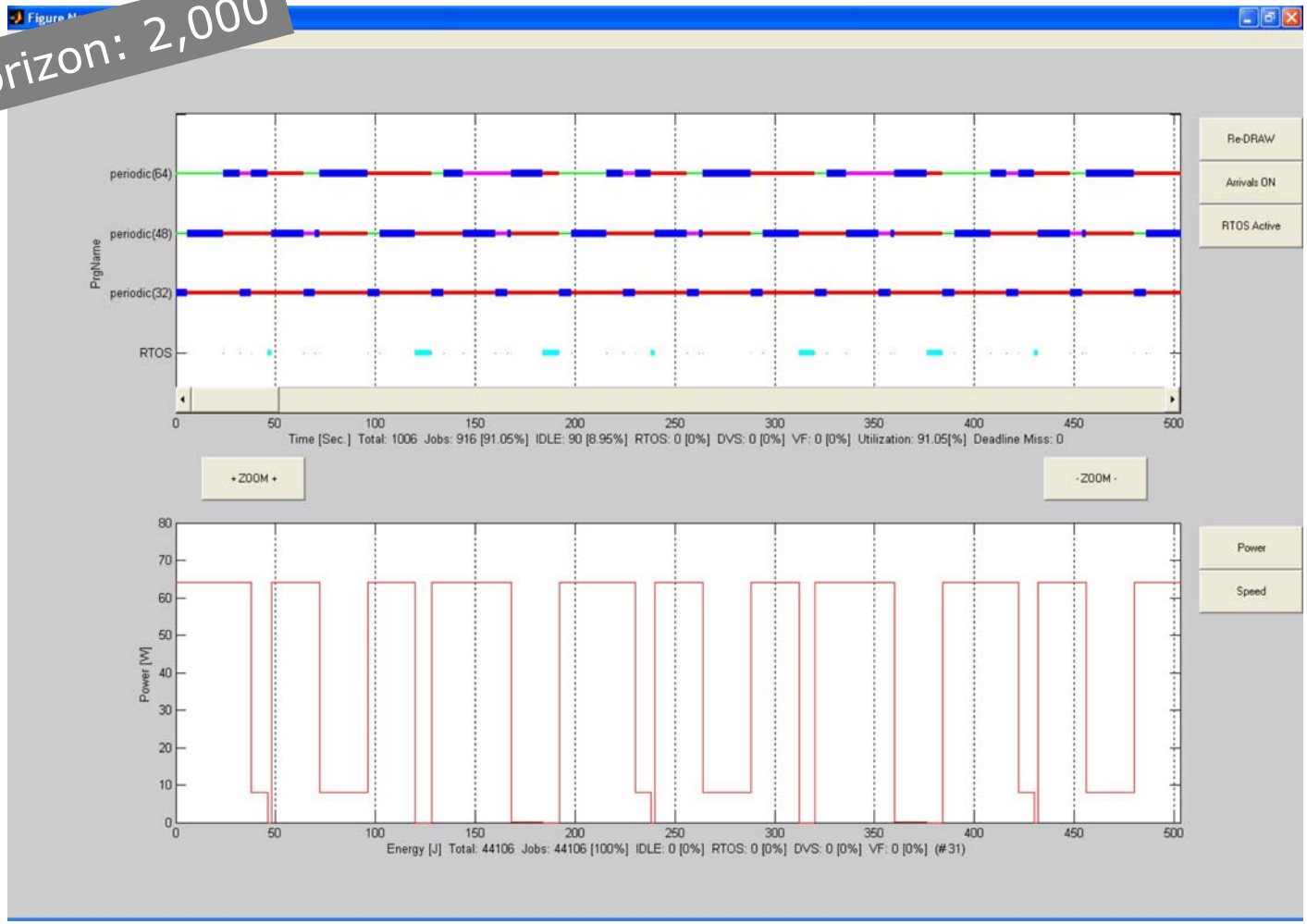


$P_1=D_1=32$ $C_1=6$
$P_2=D_2=48$ $C_2=18$
$P_3=D_3=64$ $C_3=12$

EDF w preemption no DVS: avr.: 48

Preliminary Results

Cost horizon: 2,000



$P_1=D_1=32$ $C_1=6$
$P_2=D_2=48$ $C_2=18$
$P_3=D_3=64$ $C_3=12$

EDF w preemption w DVS: avr.: 43.37

Optimal Reconfiguration of FPGA

Utilizing new features of UPPAAL 4.0

User-defined functions Types & Select

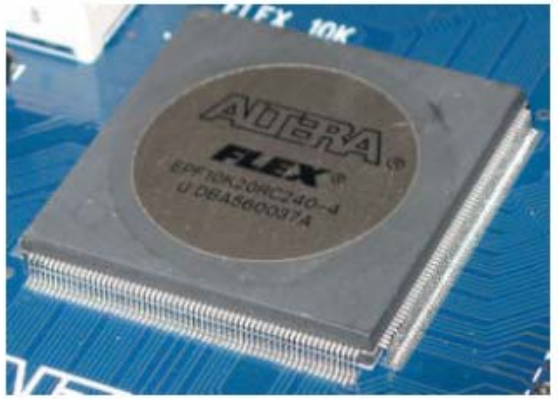
Due to Jacob I. Rasmussen

BRICS
Basic Research
in Computer Science

CISS
CENTER FOR INDLEJREDE SOFTWARE SYSTEMER

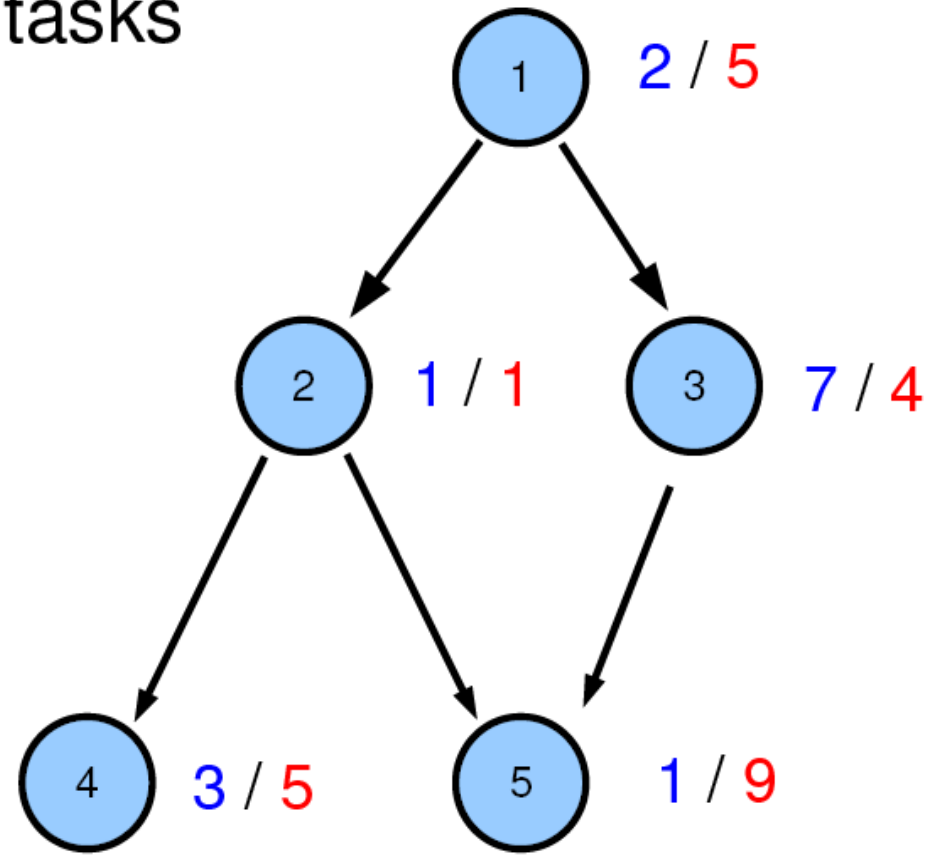
Field Programmable Gate Array

- Programmable logic device
- 2D or 1D layout
- Can mimic:
  - AND, OR, XOR, NOT gates
  - Simple flip-flops
  - etc.
- Dynamical reconfiguration



The Problem

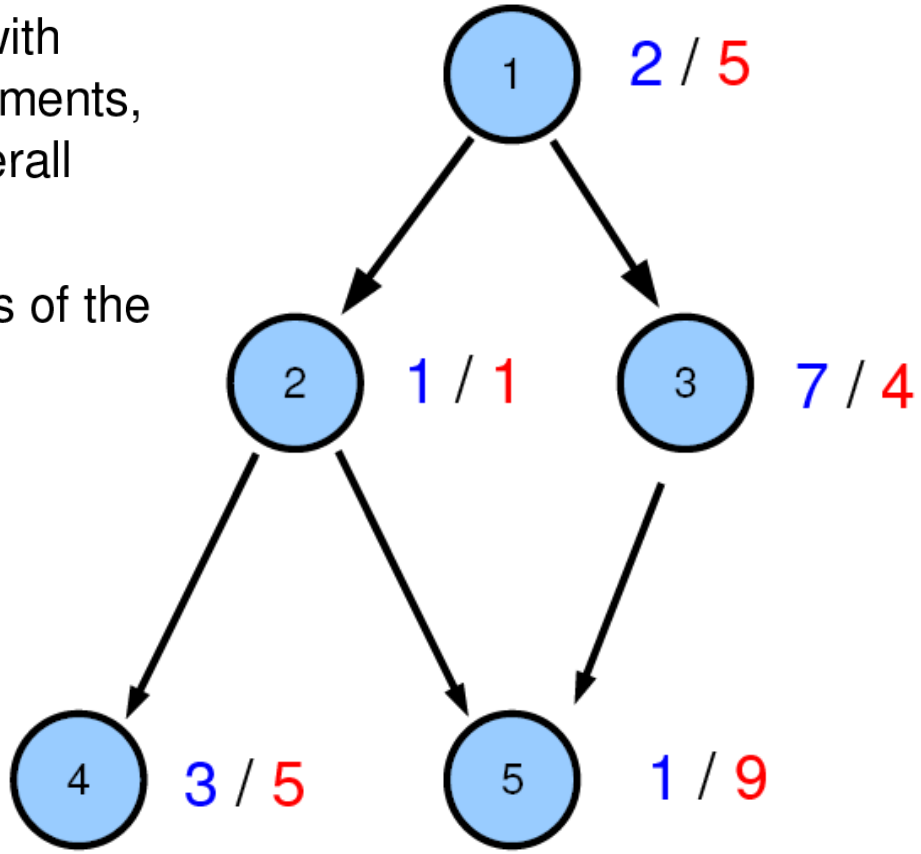
- Dependency graph of tasks
  - space requirement
  - execution time
- 1D or 2D FPGA



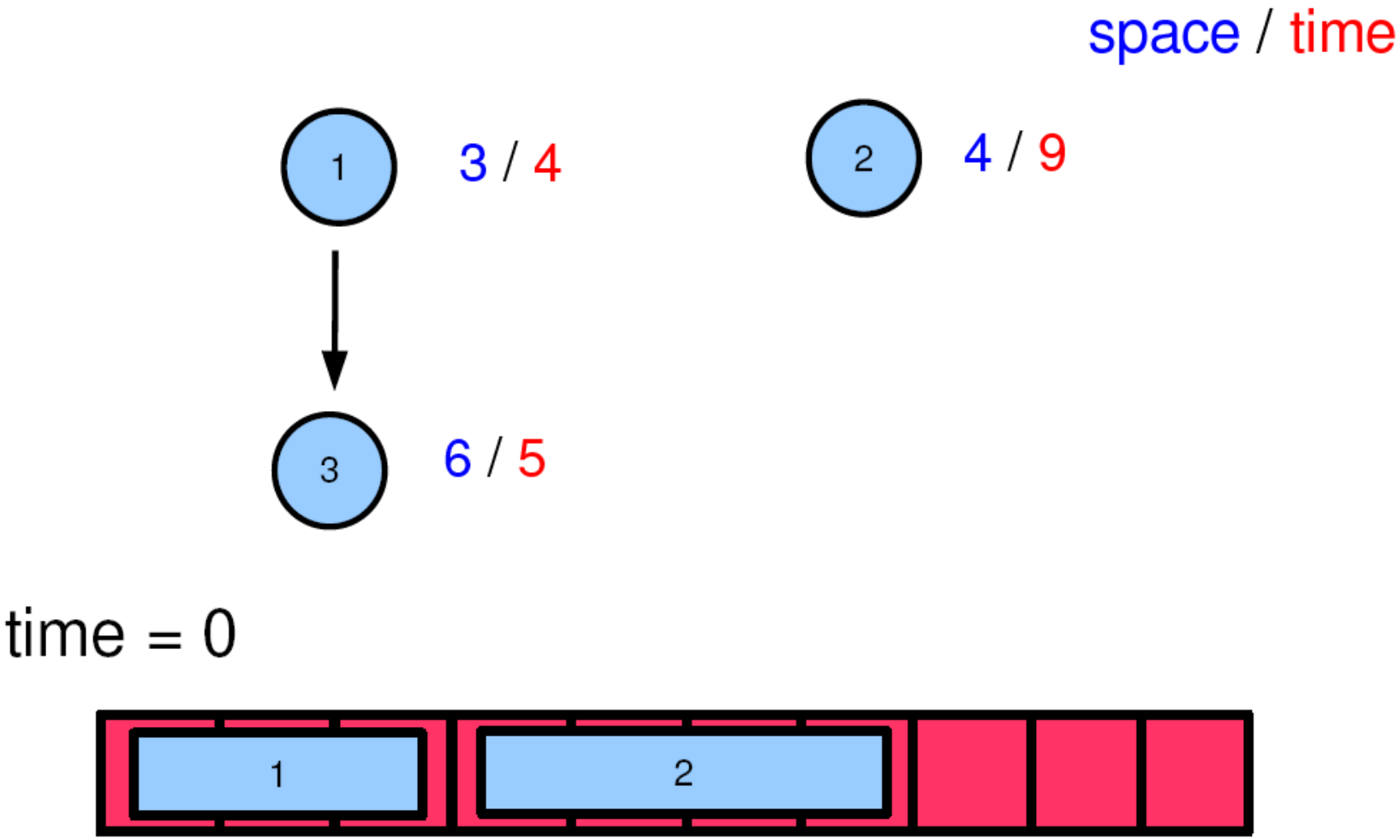
The Problem

- Given a 1-dimensional FPGA with fixed size n and a number of interdependant tasks with individual space and processing requirements, find the schedule that minimizes the overall time.
- Schedule: Collection of consistent tuples of the form:

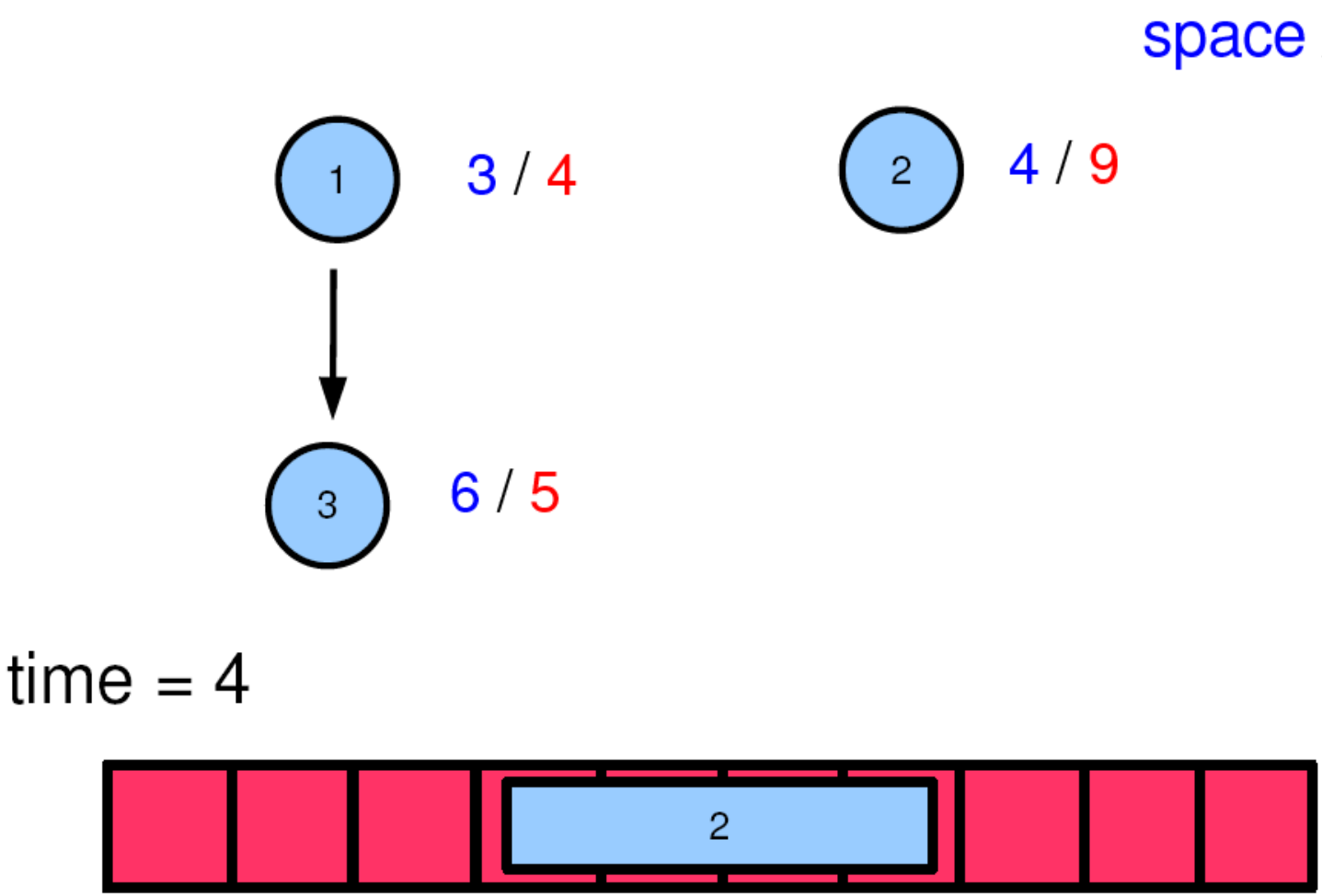
  (task_id, fpga_pos, start_time)



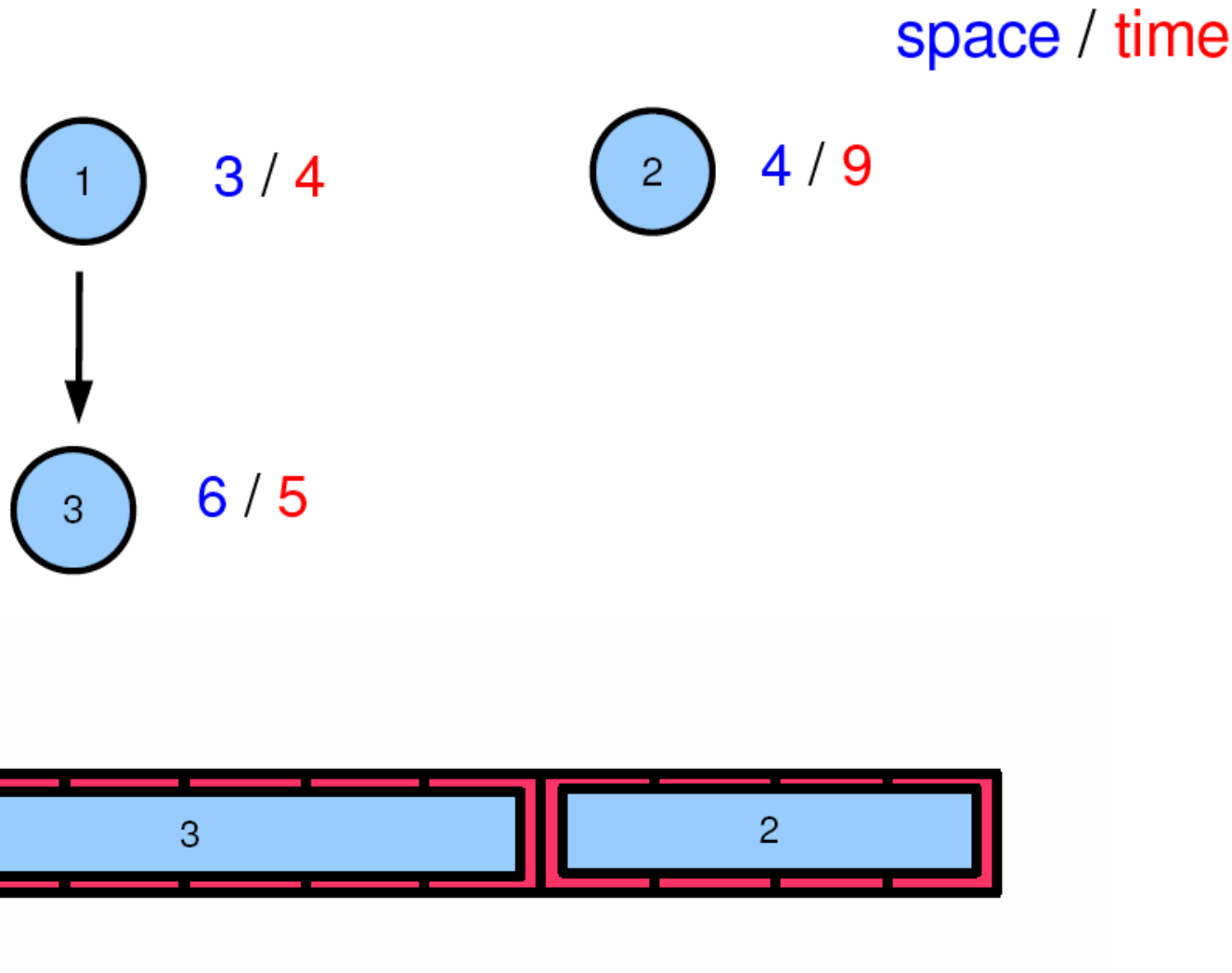
Example



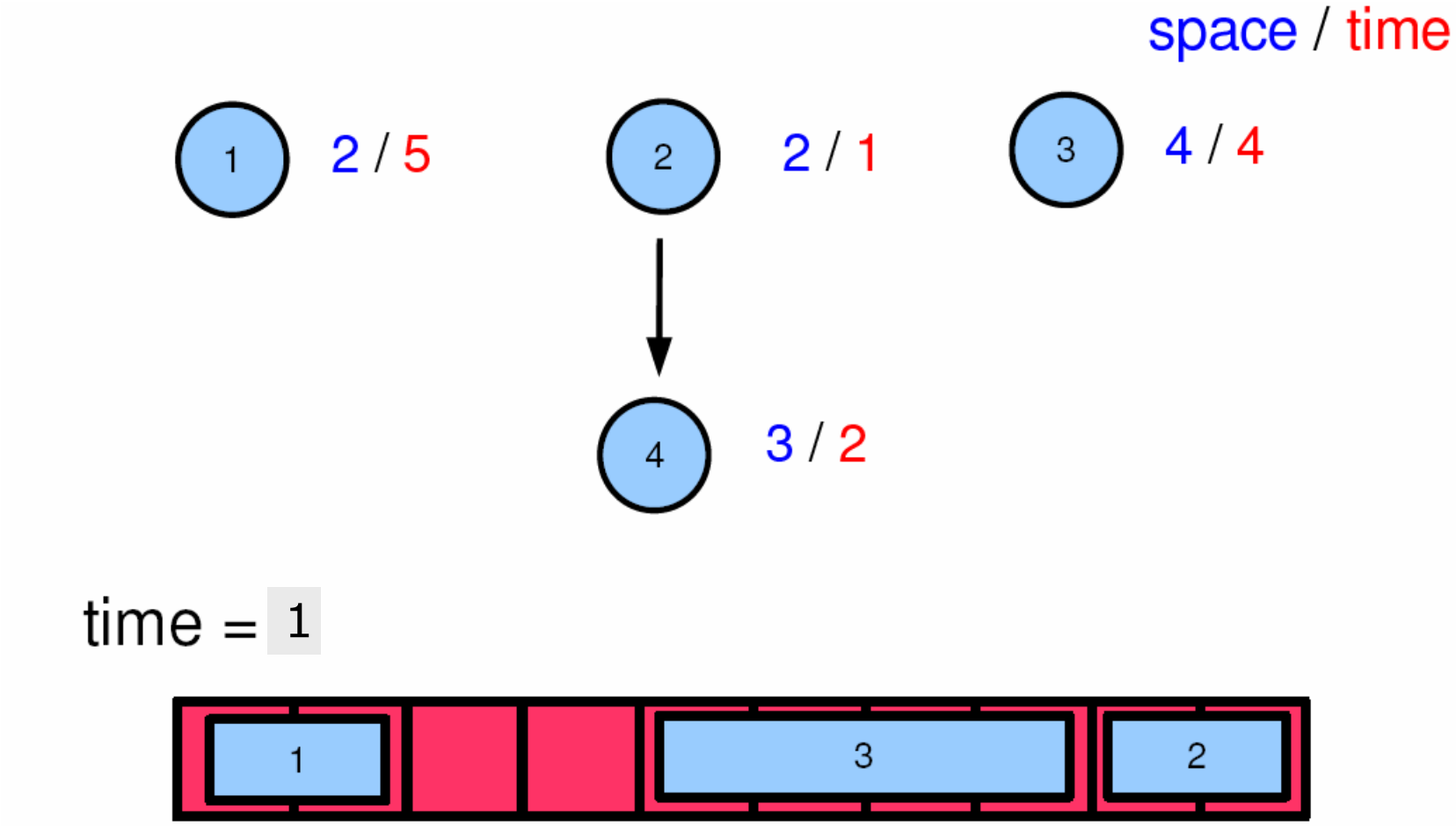
Example



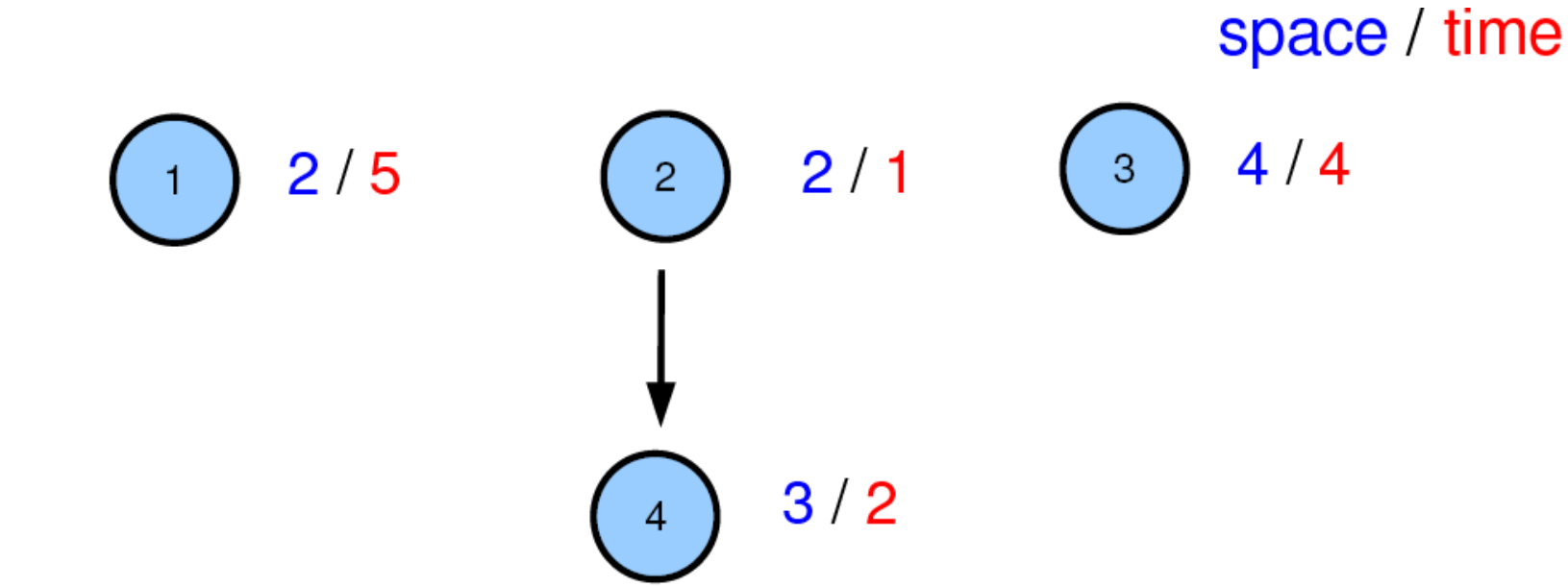
Example



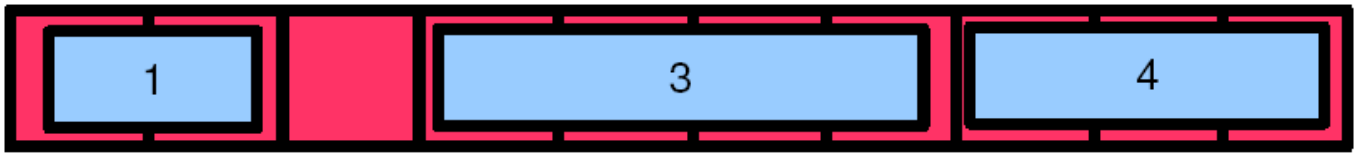
Example



Example



time = 1

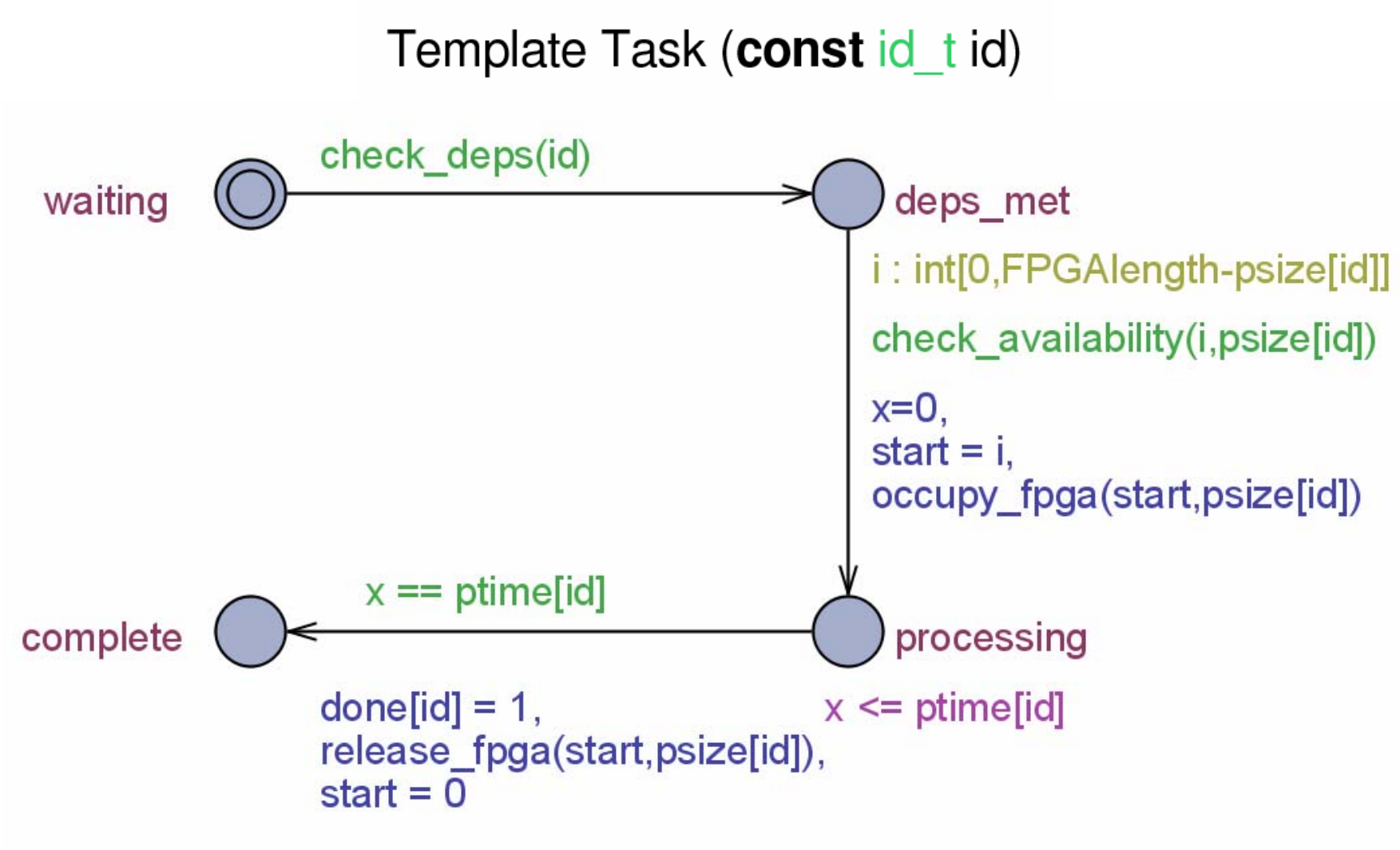


UPPAAL Model

```
const int FPGAlength = 10;
bool fpga[FPGAlength] = {0,0,0,0,0,0,0,0,0,0};
typedef int[0,FPGAlength-1] fpga_t;

const int TASKS = 4;
bool done[TASKS] = {0,0,0,0};
typedef int[0,TASKS-1] id_t;

const int psize[TASKS] = {2,2,4,3};
const int ptime[TASKS] = {5,2,4,2};

const bool deps[TASKS][TASKS] = {
{0,0,0,0}, {0,0,0,0}, {0,0,0,0}, {0,1,0,0}};
```

BRICS
Basic Research
in Computer Science

UPPAAL Model



BRICS
Basic Research
in Computer Science

UPPAAL Model

```
bool check_deps(id_t id) {
  return forall(i : id_t) deps[id][i] ? done[i] : true;
}

bool check_availability(fpga_t place, int nb) {
int i;
for (i = place; i < place+nb; i++) {
  if (fpga[i])
    return false;
}
return true;
}
```

UPPAAL Model

```
void occupy_fpga(fpga_t place, int nb) {
int i;
for (i = 0; i < nb; i++) {
  fpga[place+i] = 1;
}}

void release_fpga(fpga_t place, int nb) {
int i;
for (i = 0; i < nb; i++) {
  fpga[place+i] = 0;
}}
```

Requirement specification:
E<> **forall**(i : id_t) done[i]

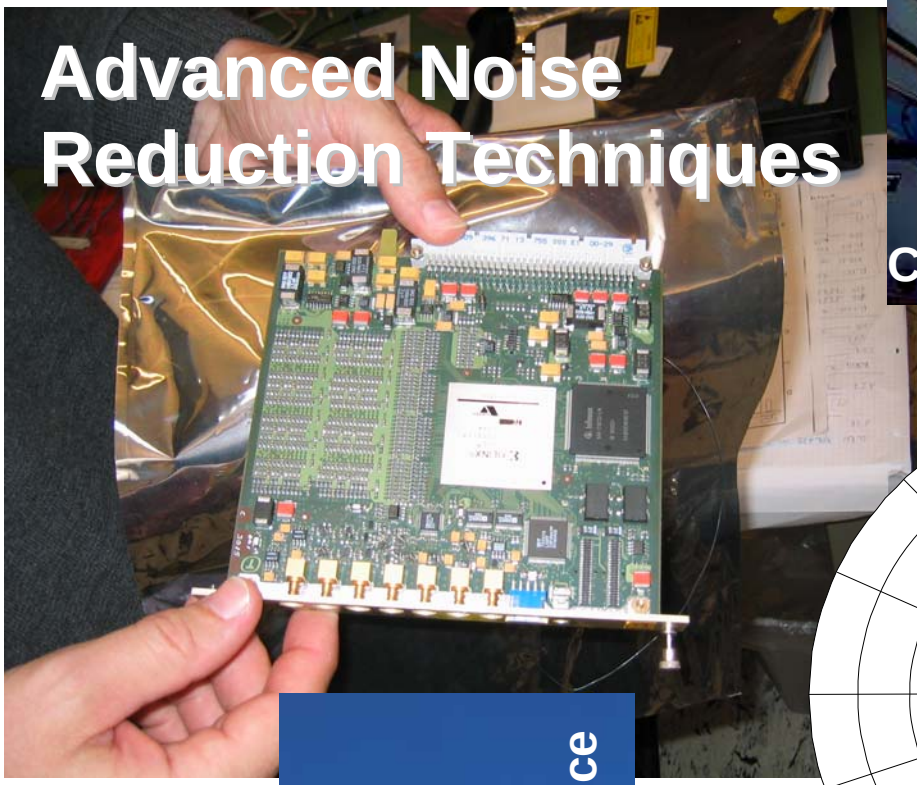
BRICS
Basic Research
in Computer Science

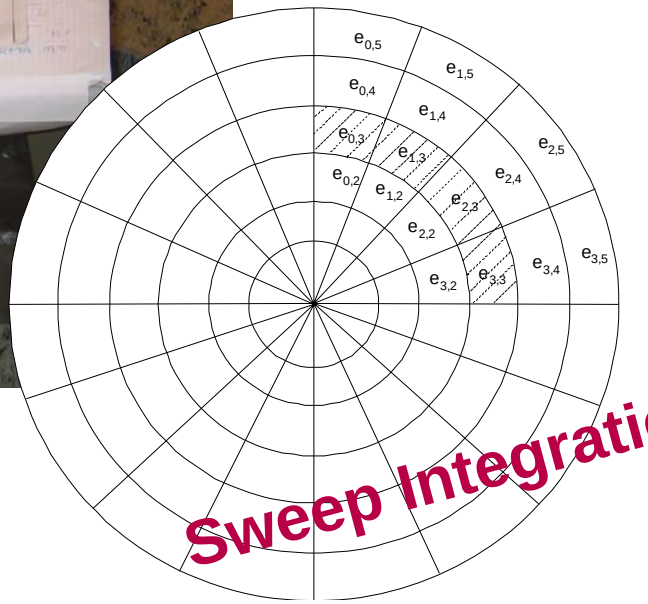
Memory Interface

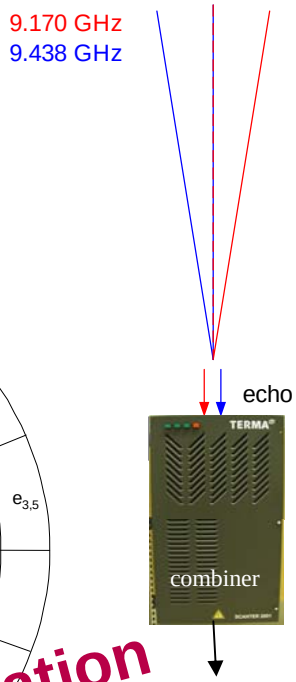
BRICS
Basic Research
in Computer Science

Memory Management

Radar Video Processing Subsystem

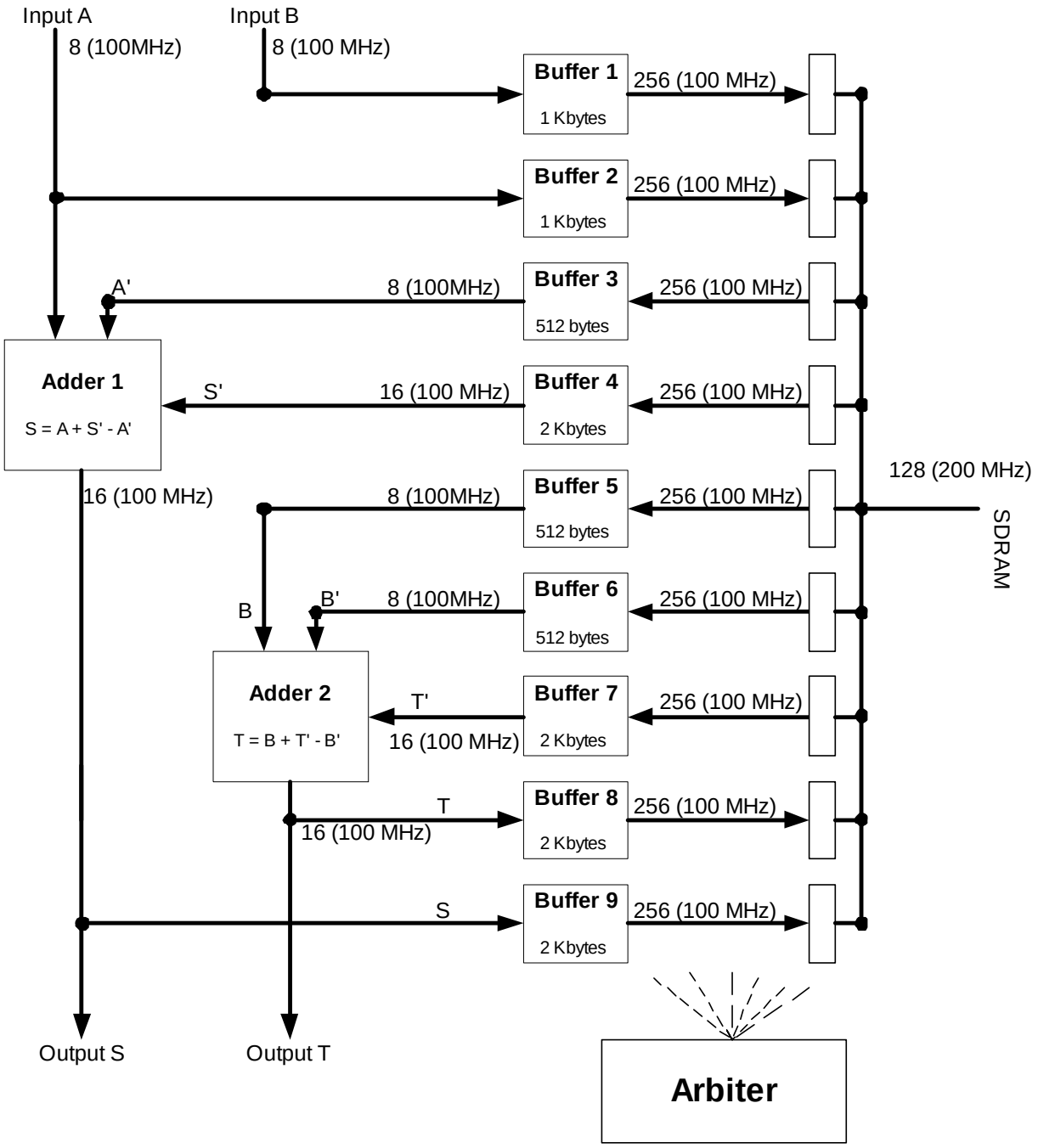


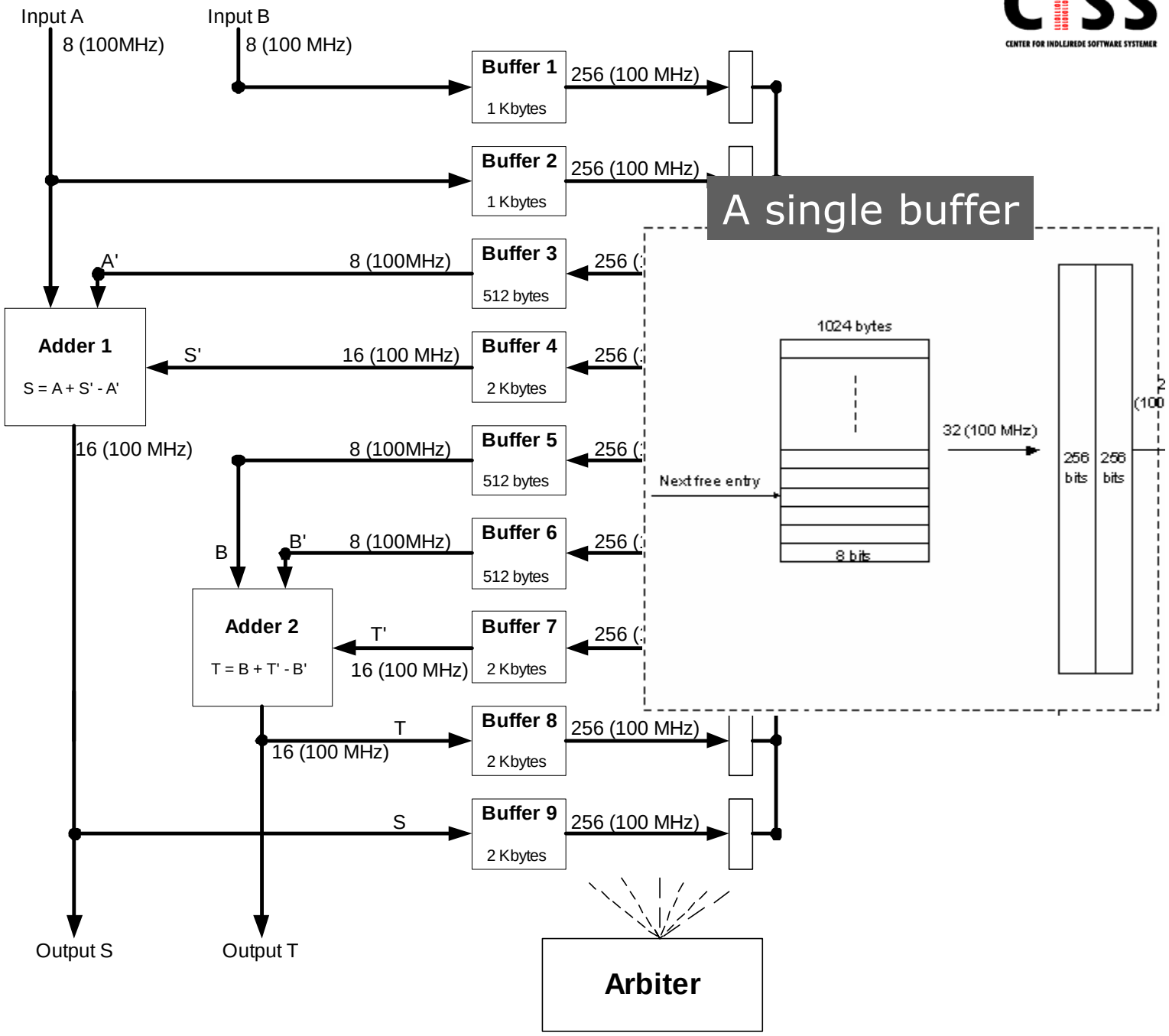


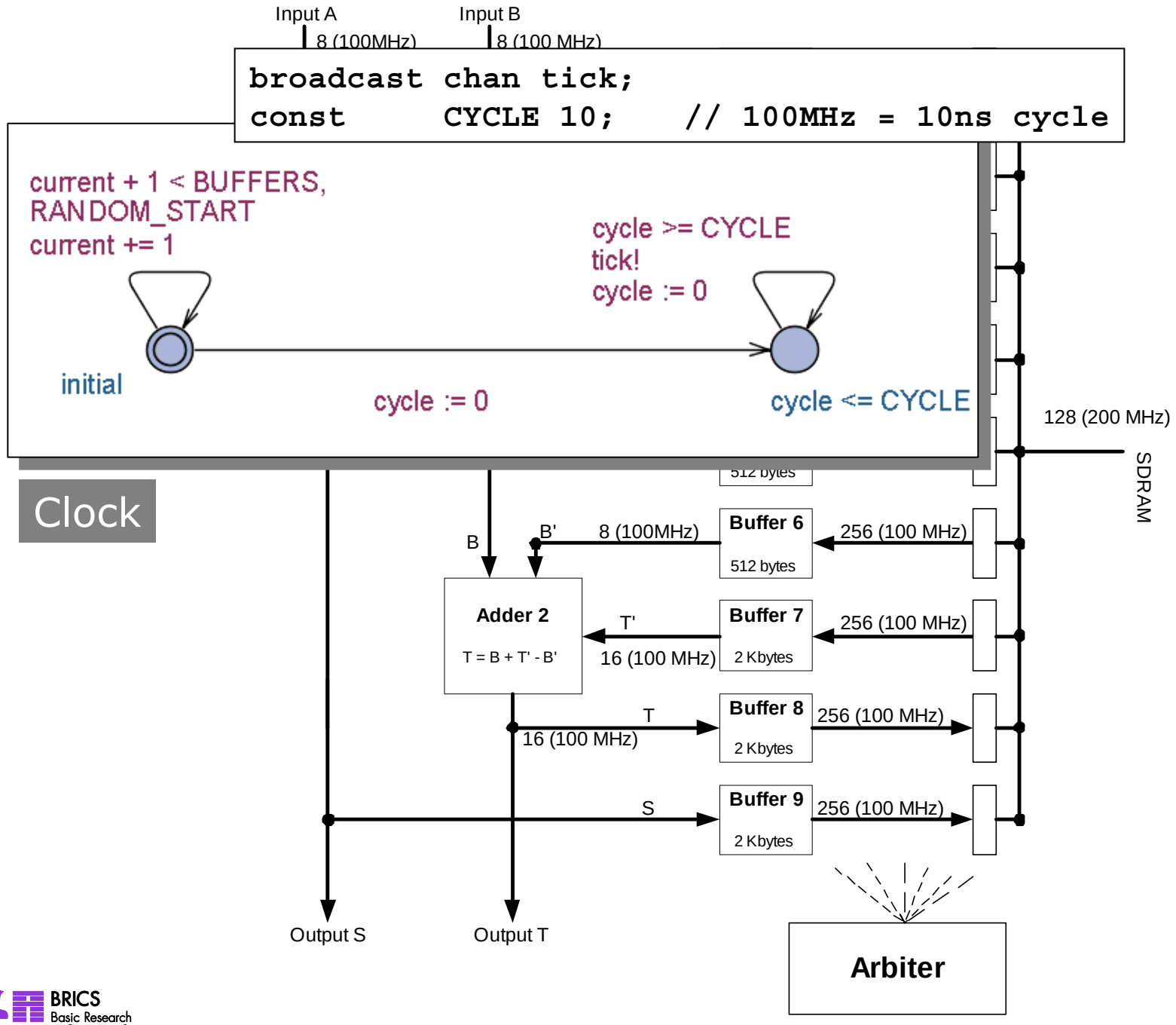


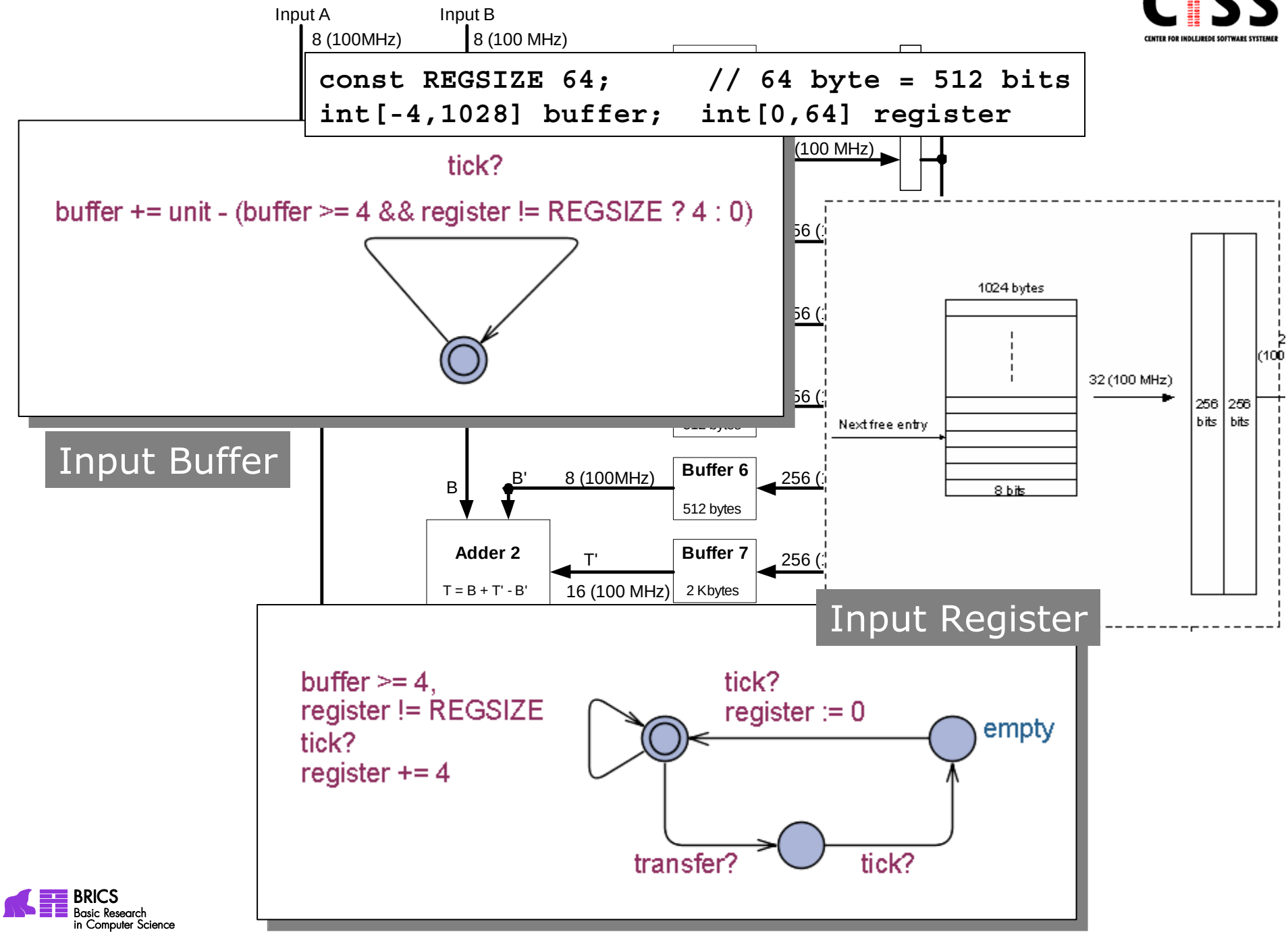
BRICS
Basic Research
in Computer Science

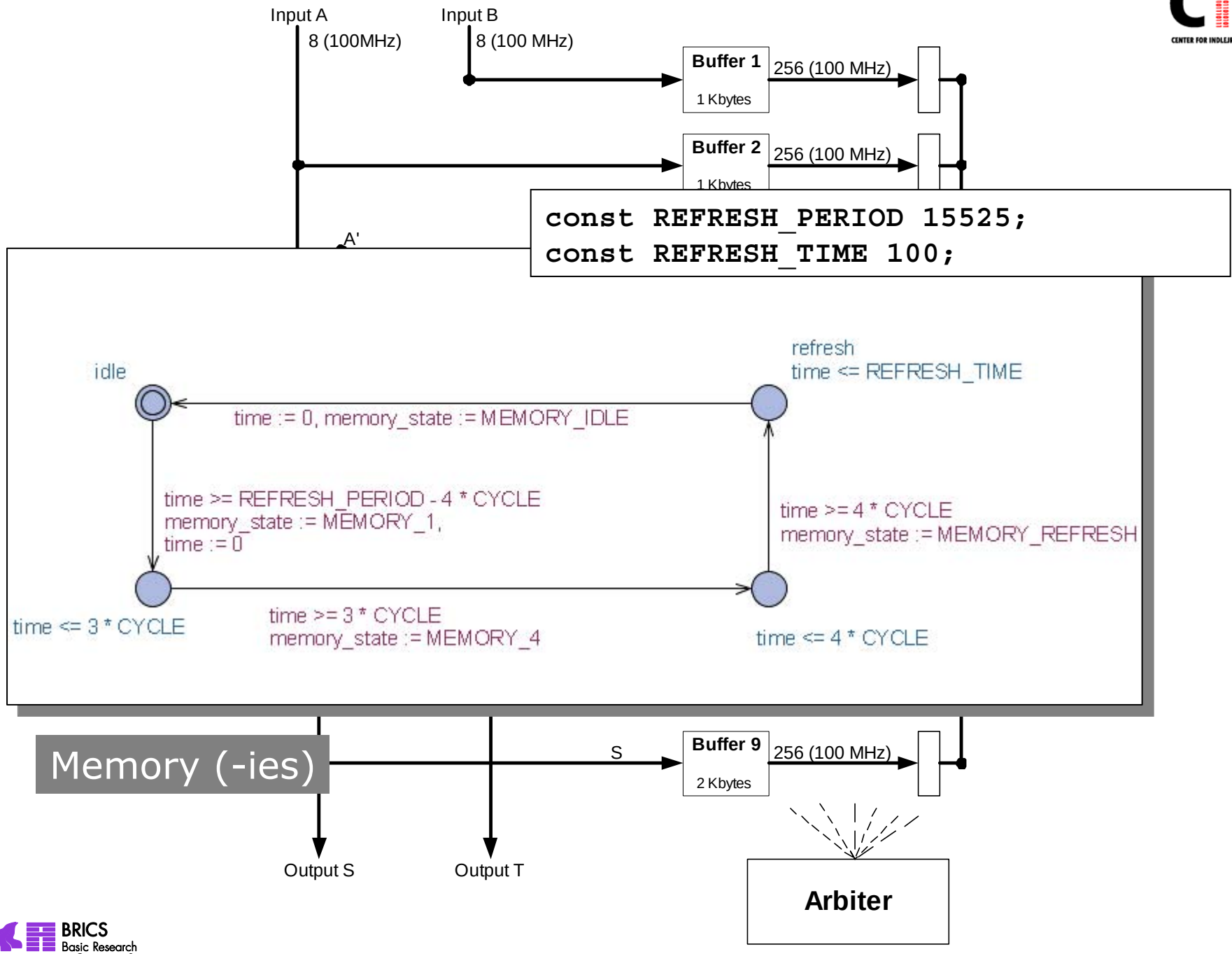
AMETIST
advanced methods for timed systems

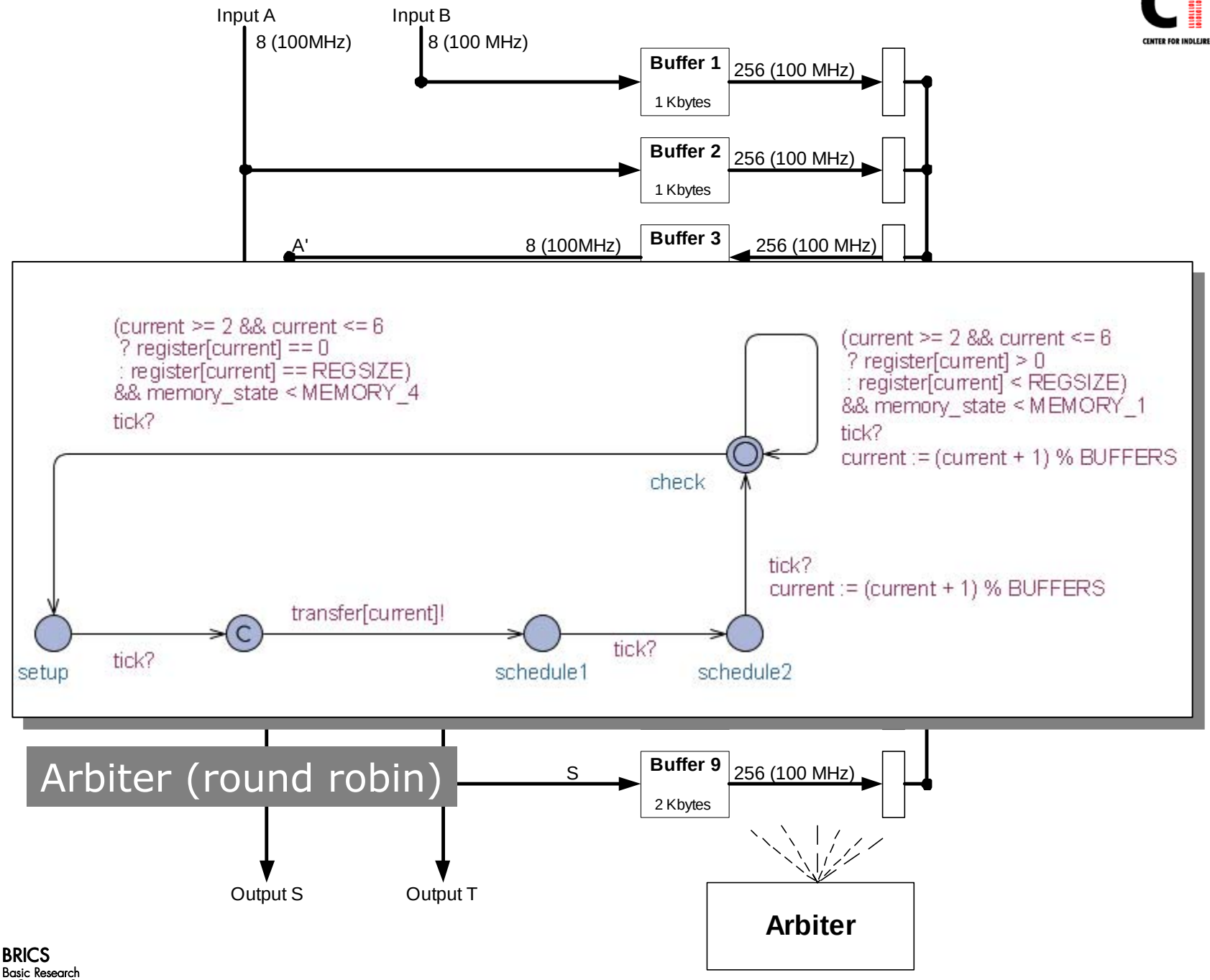

Input A
8 (100MHz)
Input B
8 (100 MHz)
Buffer 1
1 Kbytes
256 (100 MHz)
Buffer 2
1 Kbytes
256 (100 MHz)
A'
8 (100MHz)
Buffer 3
512 bytes
256 (100 MHz)
Adder 1
S = A + S' - A'
S'
16 (100 MHz)
Buffer 4
2 Kbytes
256 (100 MHz)
128 (200 MHz)
SDRAM
16 (100 MHz)
8 (100MHz)
Buffer 5
512 bytes
256 (100 MHz)
B
B'
8 (100MHz)
Buffer 6
512 bytes
256 (100 MHz)
Adder 2
T = B + T' - B'
T'
16 (100 MHz)
Buffer 7
2 Kbytes
256 (100 MHz)
T
16 (100 MHz)
Buffer 8
2 Kbytes
256 (100 MHz)
S
Buffer 9
2 Kbytes
256 (100 MHz)
Output S
Output T
Arbiter


Input A
8 (100MHz)
Input B
8 (100 MHz)
Buffer 1
1 Kbytes
256 (100 MHz)
Buffer 2
1 Kbytes
256 (100 MHz)
A single buffer
A'
8 (100MHz)
Buffer 3
512 bytes
Adder 1
S = A + S' - A'
S'
16 (100 MHz)
Buffer 4
2 Kbytes
1024 bytes
32 (100 MHz)
Next free entry
8 bits
256 bits
256 bits
16 (100 MHz)
8 (100MHz)
Buffer 5
512 bytes
B
B'
8 (100MHz)
Buffer 6
512 bytes
Adder 2
T = B + T' - B'
T'
16 (100 MHz)
Buffer 7
2 Kbytes
T
16 (100 MHz)
Buffer 8
2 Kbytes
256 (100 MHz)
S
Buffer 9
2 Kbytes
256 (100 MHz)
Output S
Output T
Arbiter


Input A
8 (100MHz)
Input B
8 (100 MHz)
broadcast chan tick;
const CYCLE 10; // 100MHz = 10ns cycle
current + 1 < BUFFERS,
RANDOM_START
current += 1
cycle >= CYCLE
tick!
cycle := 0
initial
cycle := 0
cycle <= CYCLE
Clock
128 (200 MHz)
SDRAM
512 bytes
B
B'
8 (100MHz)
Buffer 6
512 bytes
256 (100 MHz)
Adder 2
T = B + T' - B'
T'
16 (100 MHz)
Buffer 7
2 Kbytes
256 (100 MHz)
T
16 (100 MHz)
Buffer 8
2 Kbytes
256 (100 MHz)
S
Buffer 9
2 Kbytes
256 (100 MHz)
Output S
Output T
Arbiter


Input A
8 (100MHz)
Input B
8 (100 MHz)
const REGSIZE 64; // 64 byte = 512 bits
int[-4,1028] buffer; int[0,64] register
(100 MHz)
tick?
buffer += unit - (buffer >= 4 && register != REGSIZE ? 4 : 0)
Input Buffer
1024 bytes
32 (100 MHz)
Next free entry
8 bits
256 bits
256 bits
B
B'
8 (100MHz)
Buffer 6
512 bytes
Adder 2
T = B + T' - B'
T'
16 (100 MHz)
Buffer 7
2 Kbytes
Input Register
buffer >= 4,
register != REGSIZE
tick?
register += 4
tick?
register := 0
empty
transfer?
tick?


Input A
8 (100MHz)
Input B
8 (100 MHz)
Buffer 1
1 Kbytes
256 (100 MHz)
Buffer 2
1 Kbytes
256 (100 MHz)
const REFRESH_PERIOD 15525;
const REFRESH_TIME 100;
A'
idle
refresh
time <= REFRESH_TIME
time := 0, memory_state := MEMORY_IDLE
time >= REFRESH_PERIOD - 4 * CYCLE
memory_state := MEMORY_1,
time := 0
time >= 4 * CYCLE
memory_state := MEMORY_REFRESH
time <= 3 * CYCLE
time >= 3 * CYCLE
memory_state := MEMORY_4
time <= 4 * CYCLE
Memory (-ies)
S
Buffer 9
2 Kbytes
256 (100 MHz)
Output S
Output T
Arbiter


Input A
8 (100MHz)
Input B
8 (100 MHz)
Buffer 1
1 Kbytes
256 (100 MHz)
Buffer 2
1 Kbytes
256 (100 MHz)
A'
8 (100MHz)
Buffer 3
256 (100 MHz)
(current >= 2 && current <= 6
? register[current] == 0
: register[current] == REGSIZE)
&& memory_state < MEMORY_4
tick?
(current >= 2 && current <= 6
? register[current] > 0
: register[current] < REGSIZE)
&& memory_state < MEMORY_1
tick?
current := (current + 1) % BUFFERS
check
tick?
current := (current + 1) % BUFFERS
transfer[current]!
setup
tick?
C
schedule1
tick?
schedule2
Arbiter (round robin)
S
Buffer 9
2 Kbytes
256 (100 MHz)
Output S
Output T
Arbiter