DRT defense, December 19th 2007

Yussef Bouzouzou

Parallel Simulation of Systems-on-Chip Models at the Transaction Level

JURY members:

Christian Boitet	President	(Clips-UJF)
Jean-Luc Dekeyser	Reviewer	(Lifl-LILLE)
Frédéric Pétrot	Reviewer	(Tima-INPG)
Laurent Maillet-Contoz	Examiner	(STMicroelectronics)
Florence Maraninchi	Advisor	(Verimag-INPG)
Pascal Raymond	Co-advisor	(Verimag-CNRS)
Emmanuel Dufour	Advisor	(Silicomp-Orange)

orange Business Services

Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

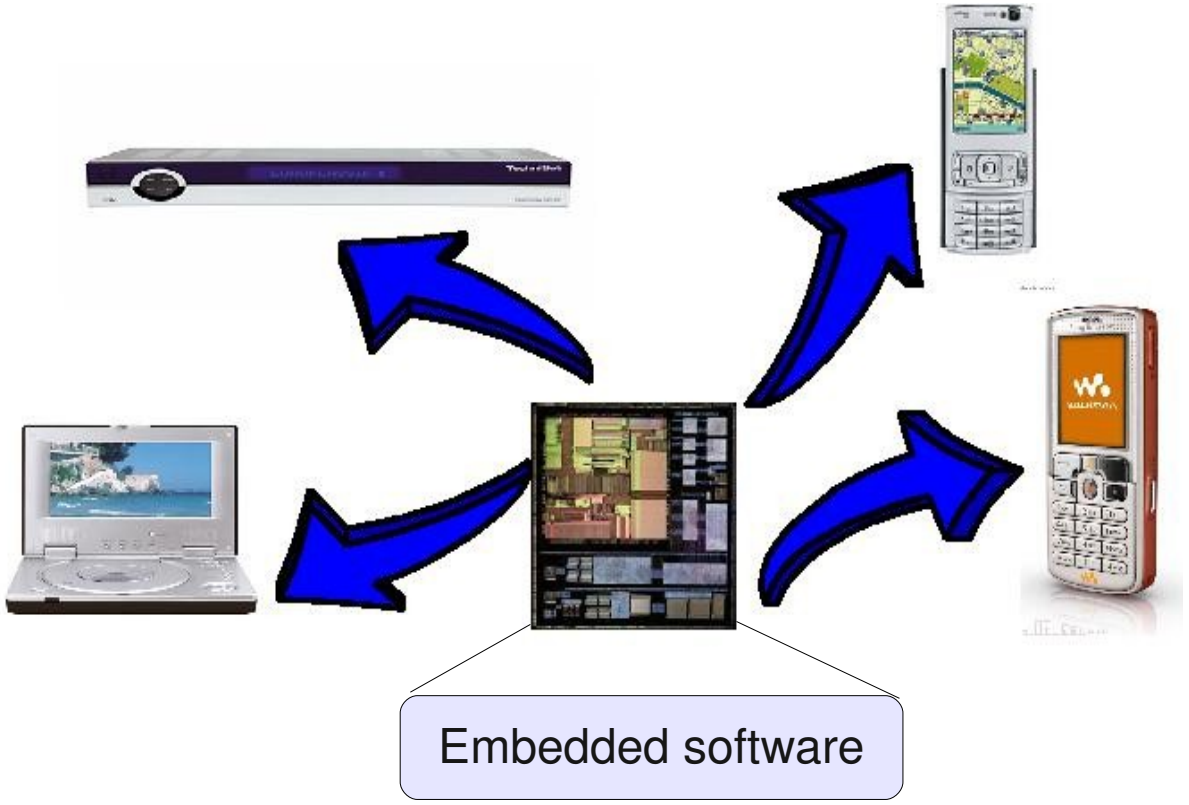
Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work

Systems on Chip (SoC)



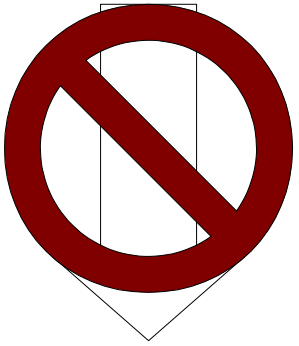
Hardware components (IP) : processor(s), memories, DMA (Direct Memory Access), bus ...
+ embedded software (OS + Applications)

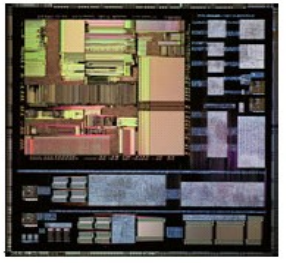
Testing the embedded software on the physical chip

Embedded software

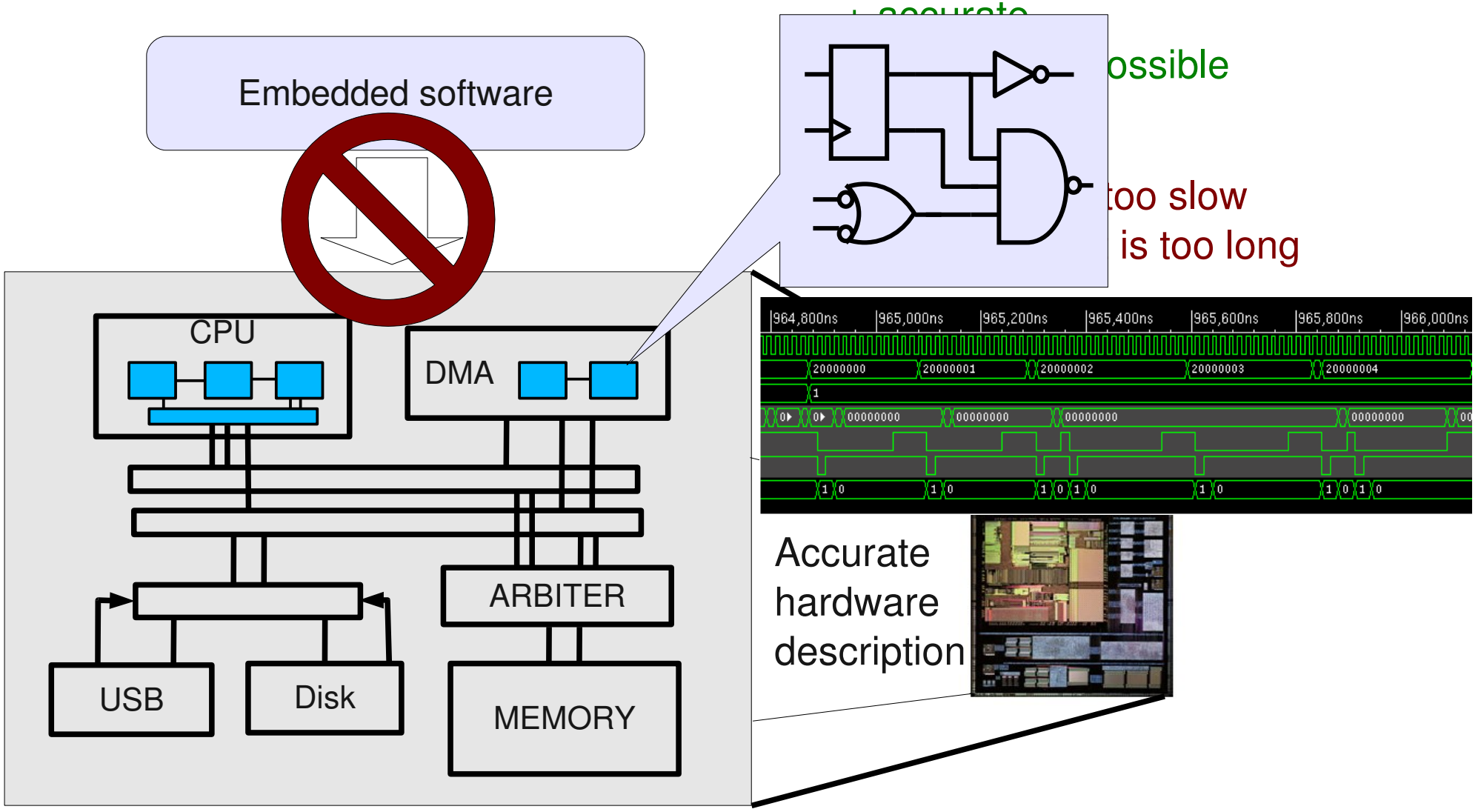
+ fast
+ accurate

- « black box »
- bug => high cost





Simulating the embedded software on an accurate hardware description

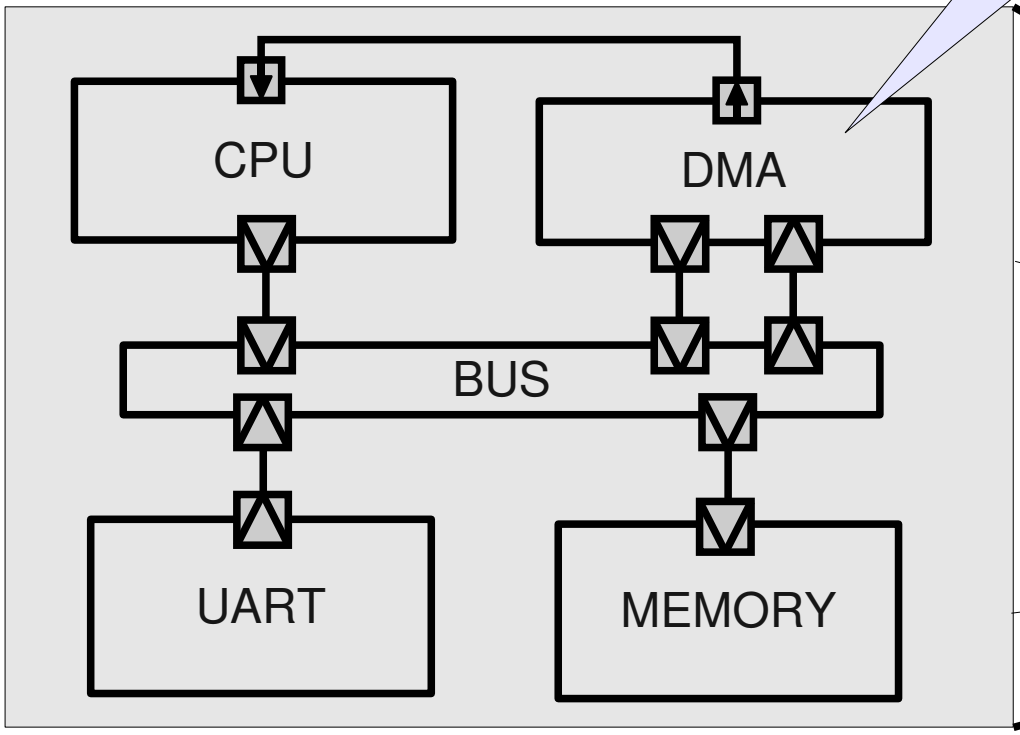


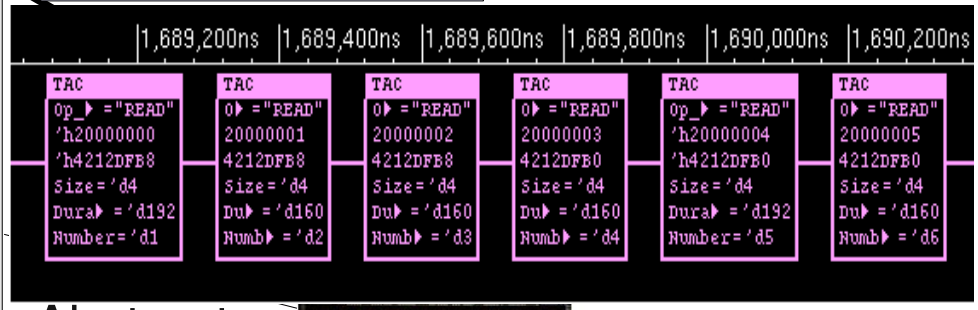
Simulating the embedded software on an abstract hardware model

Embedded software

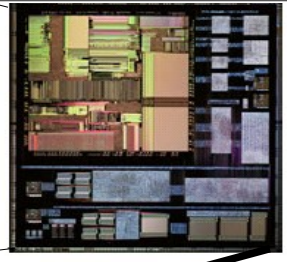
D = read(A);
transform(D);
write(D, A);

+ software co-design
+ nesis



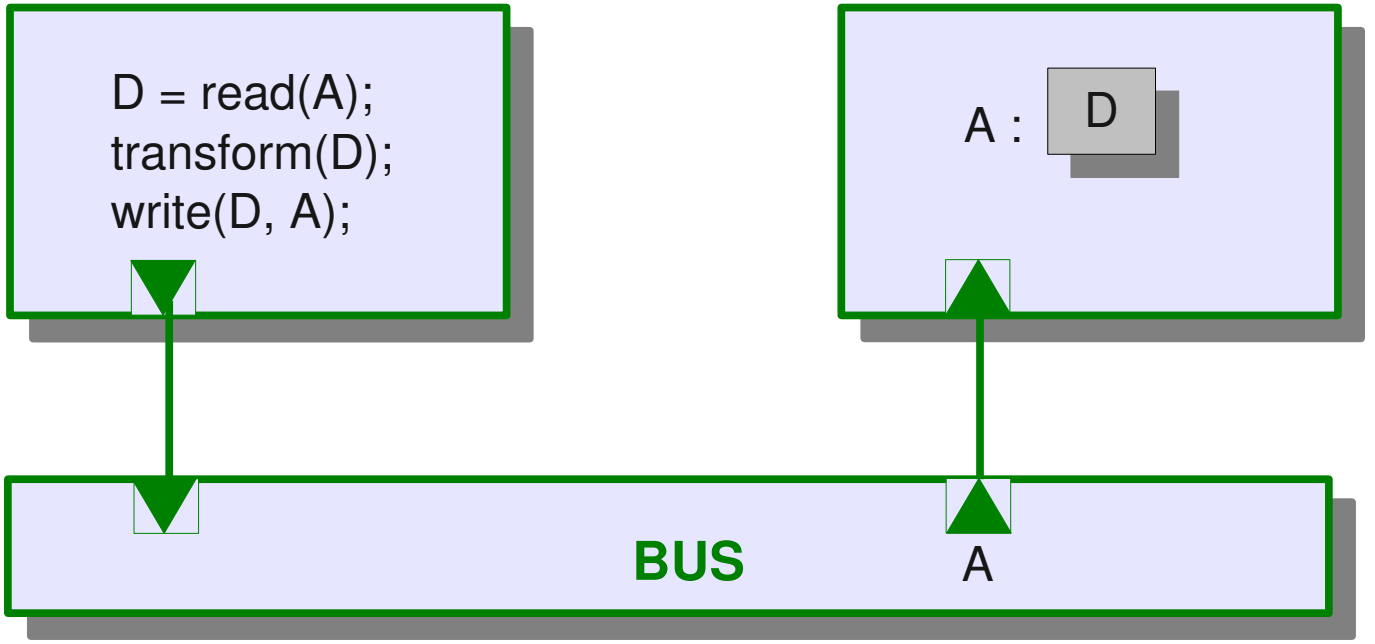


Abstract model of the hardware



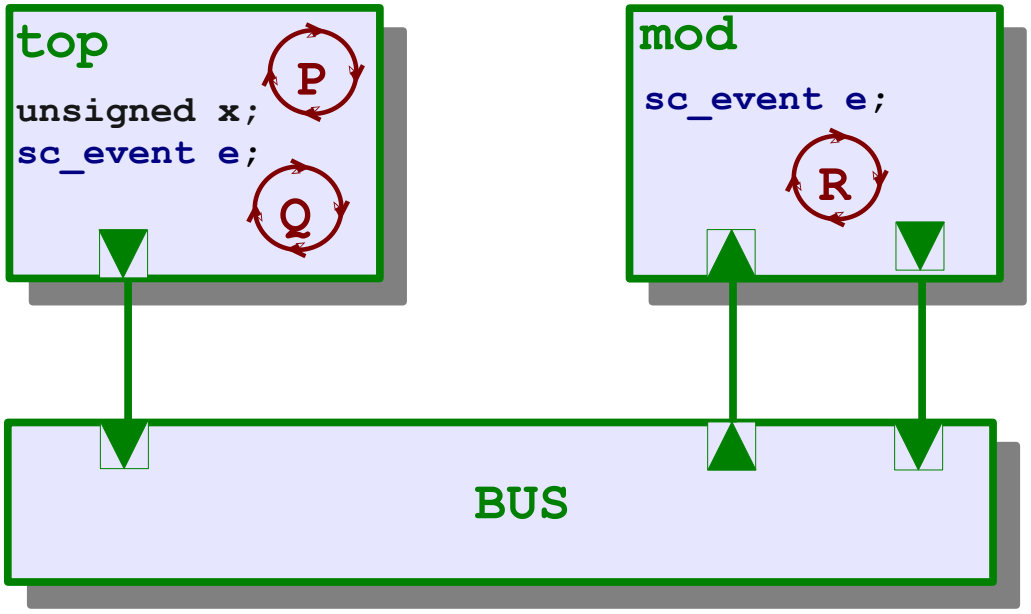
Transactional Model (TLM) : principle

transaction = atomic exchange of data between modules
A concept of **port** (component interface)
a **bus** (transport addresses and data)
a **communication protocol** (e.g., read(), write())



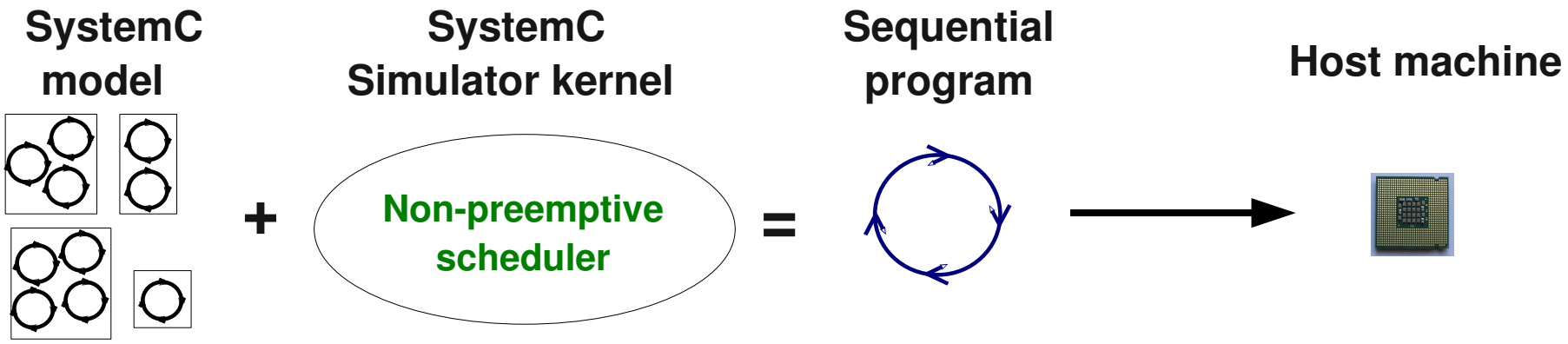
SystemC : C++ library for the modeling of embedded systems

```
SC_MODULE(top){
  unsigned x;
  sc_event e;
  SC_HAS_PROCESS(top);
  SC_CTOR(top){
    bus.port(M.port);
    SC_THREAD(P);
    SC_THREAD(Q);
  }
  void top::P() {
    ... wait(e); ... }
}
```



The whole C++ language, plus :
modules and *ports* classes describing the architecture,
Processes and ***events*** describing the behavior,
A concept of simulation time,
A simulation engine : non-preemptive (collaborative) scheduler

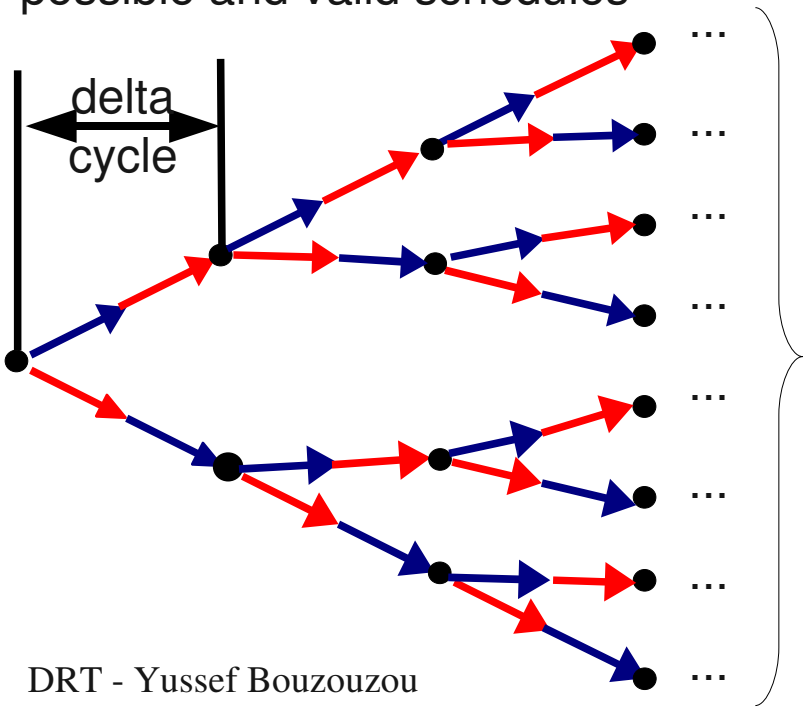
Official execution semantics



non-preemptive : SystemC processes decide when to yield back to the scheduler

non-determinist norm : many possible and valid schedules

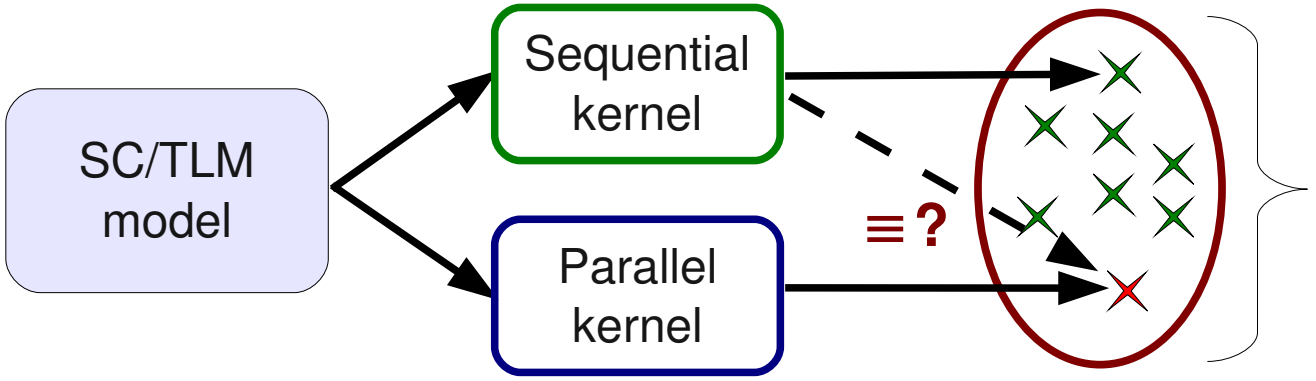
```
void top::P(){
   while(true){
      cout << "p" <<  endl;
      wait(2, SC_SEC);
   }
}
void top::Q(){
   while(true){
      cout << "q" << endl;
      wait(2, SC_SEC);
   }
}
```



Set of valid behaviors (wrt the IEEE standard)

Objective

Speed up SystemC/TLM simualtions
on multiprocessors computers (SMP)
Do not change the semantics (IEEE-1666 standard)
Without manual patches nor annotations of the model



Behaviors allowed by the IEEE standard

≡? : we need to decide
What we want to compare
When we compare

Semantics-preserving parallel execution

Comparison critera ≡ :

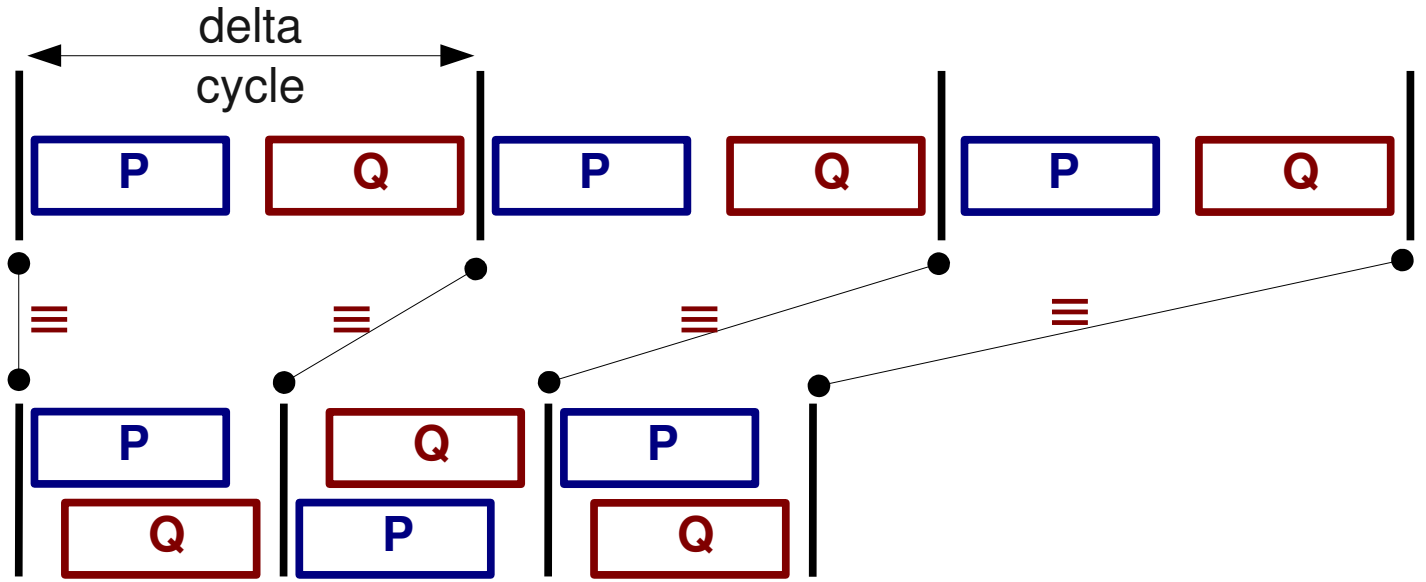
The same outputs on the screen,

The memory state (= variable values)

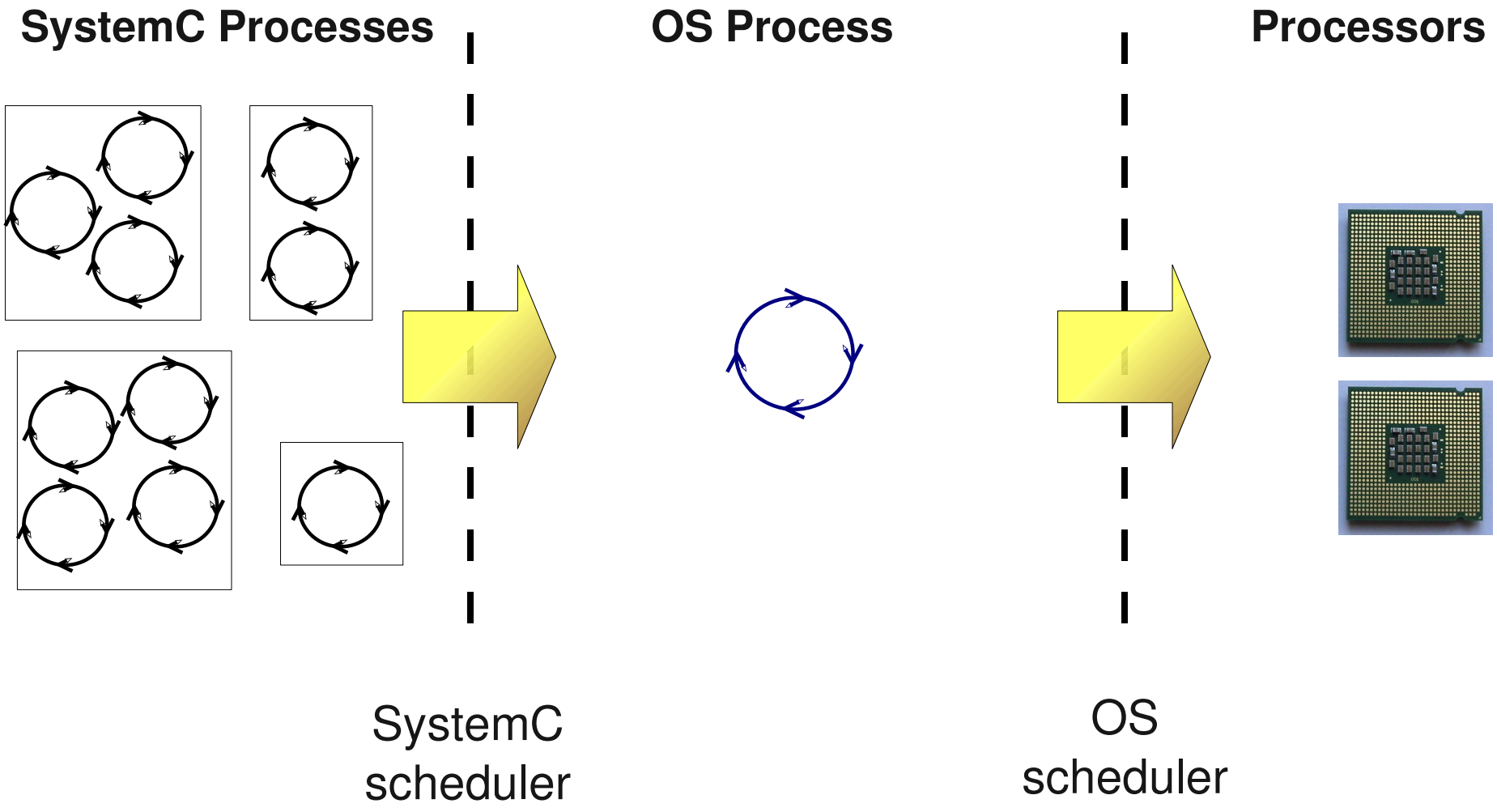
When?:

always (asm instruction): no way to parallelize

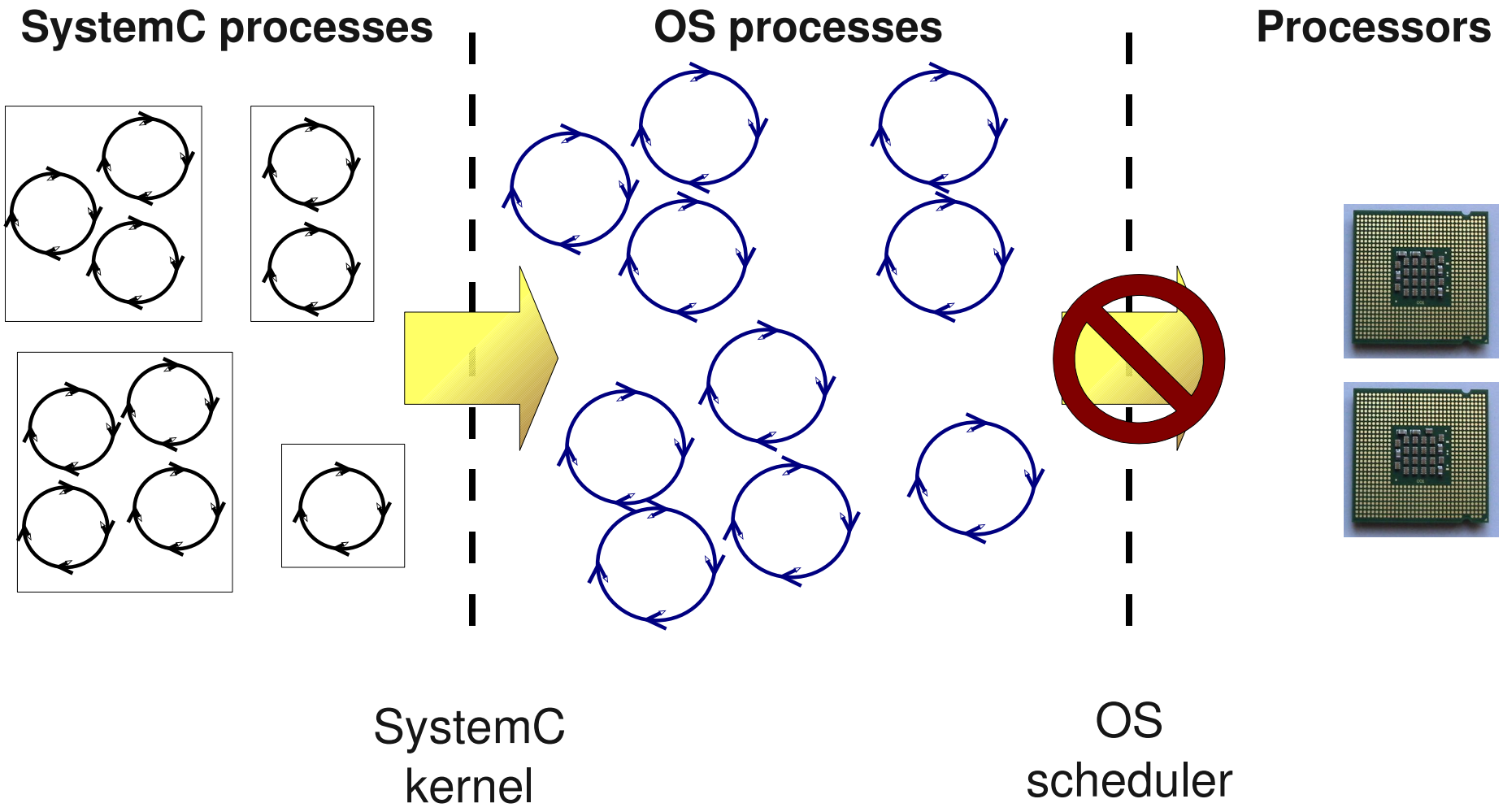
At delta-cycle borders:



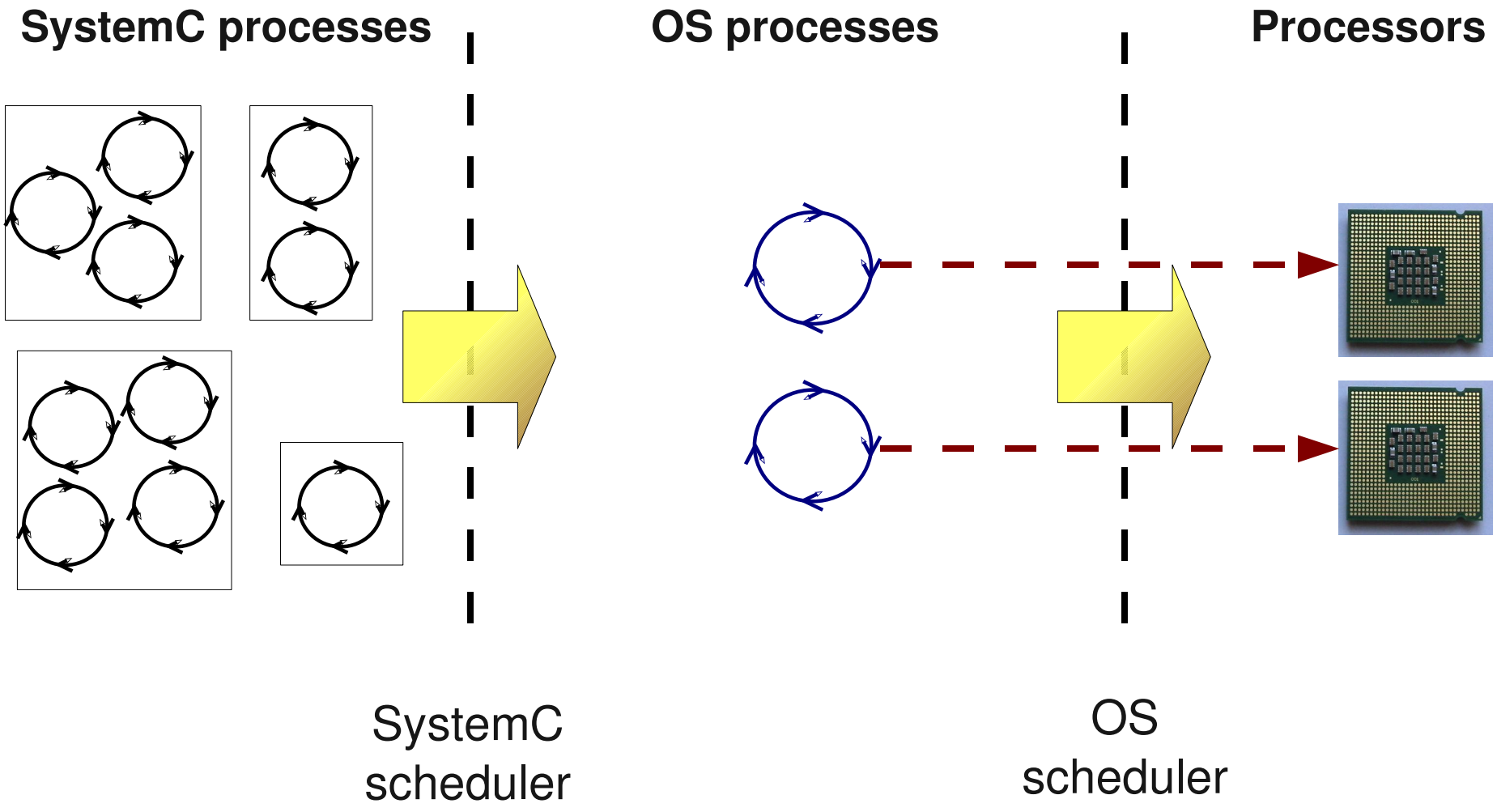
Using multiprocessor machines



Using multiprocessor machines



Using multiprocessor machines



Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

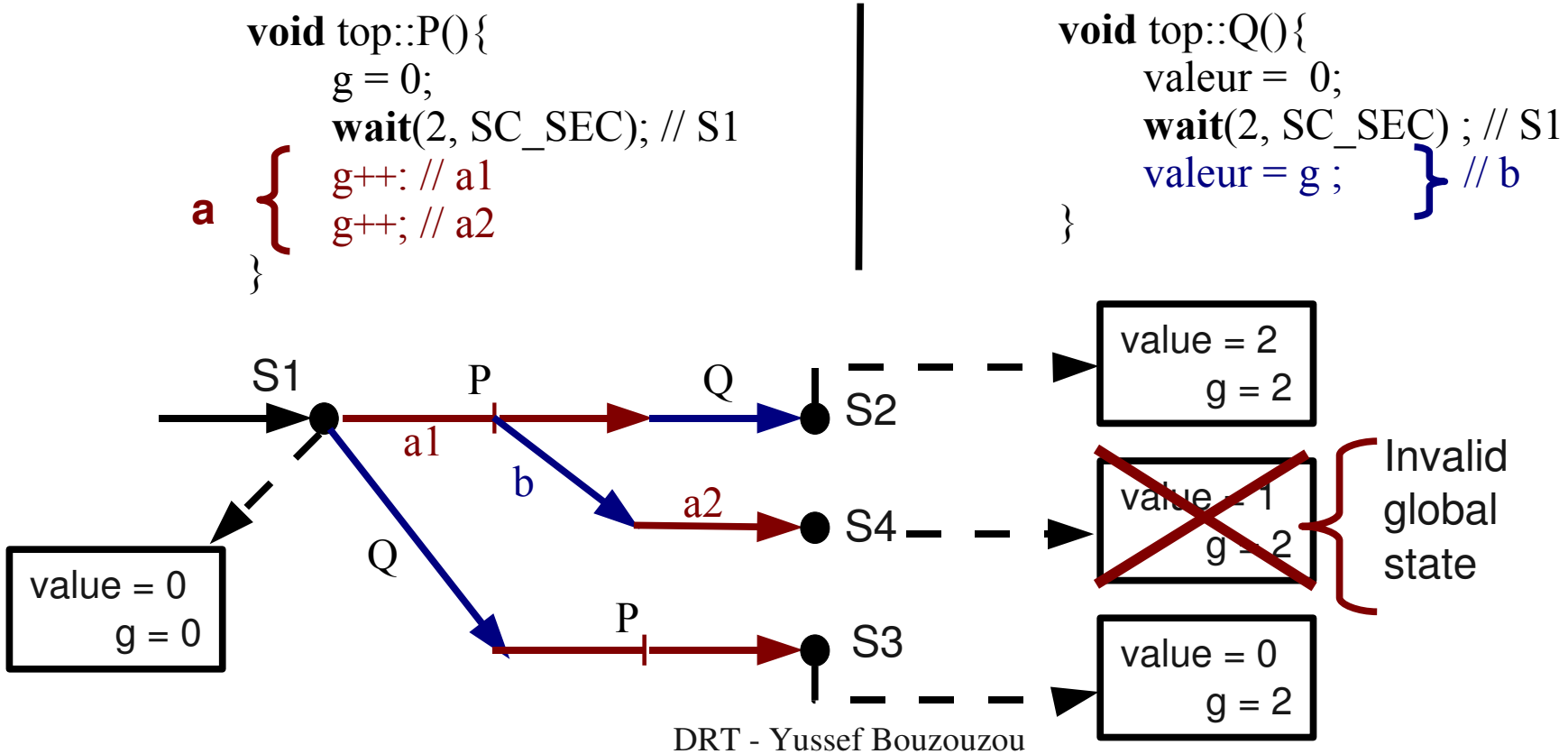
Algorithm and implementation

Experiments and results

Conclusion and further work

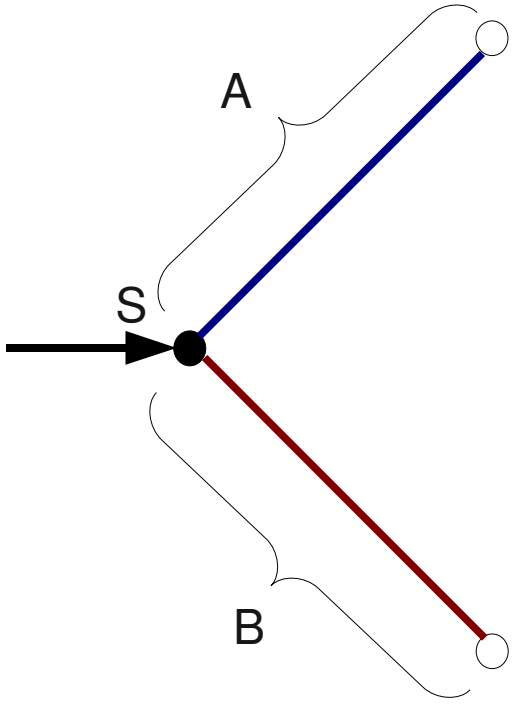
Transition, action, and interleaving

Transition : smallest atomic section w.r.t. SystemC
Action : smalleset atomic section w.r.t. the OS
a||b : the interleavings of the actions a and b



Independent tasks:
consequences for parallel executions

S is a state. A and B re two tasks to be executed from S.
A and B are **independent** if they do not share **resources**
consequence : all interleavings lead to a same state S'
A and B are independent => A || B is valid, w.r.t. the IEEE standard



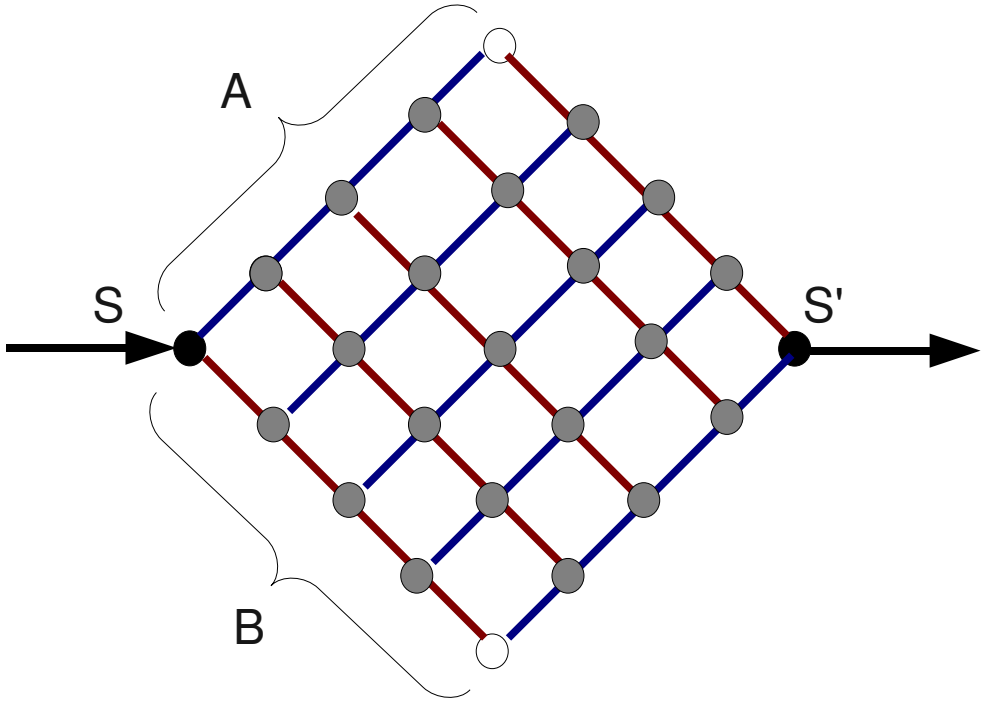
Independent tasks:
consequences for parallel executions

S is a state. A and B re two tasks to be executed from S.

A and B are **independent** if they do not share **resources**

consequence : all interleavings lead to a same state S'

A and B are independent => A || B is valid, w.r.t. the IEEE standard



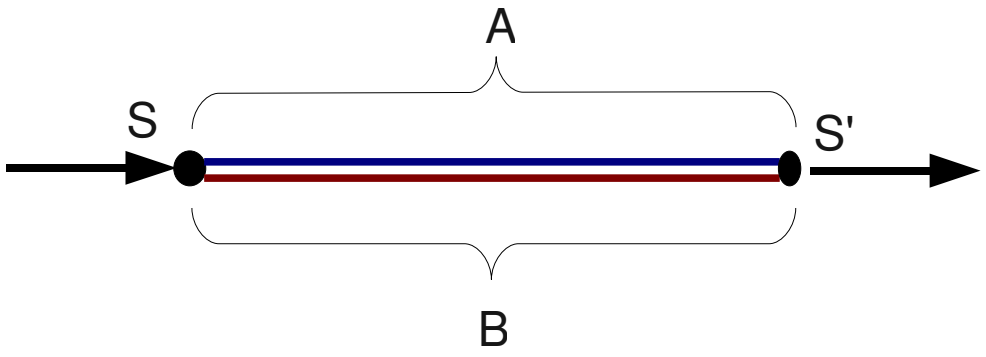
Independent tasks:
consequences for parallel executions

S is a state. A and B re two tasks to be executed from S.

A and B are **independent** if they do not share **resources**

consequence : all interleavings lead to a same state S'

A and B are independent => A || B is valid, w.r.t. the IEEE standard

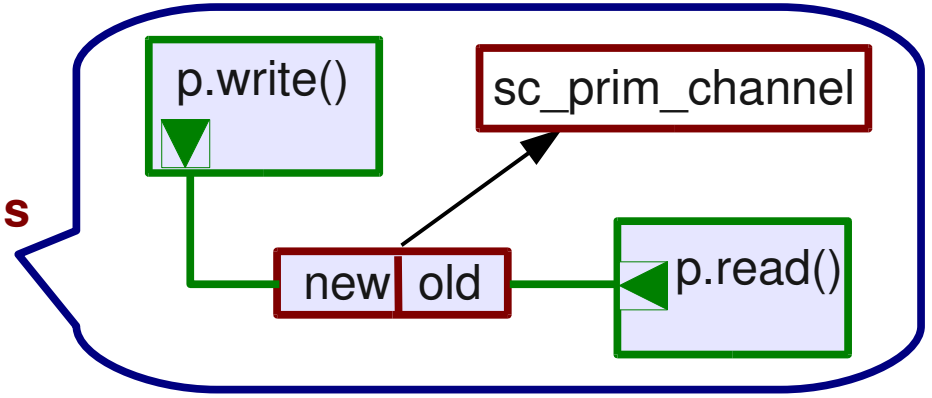


Task structural independance

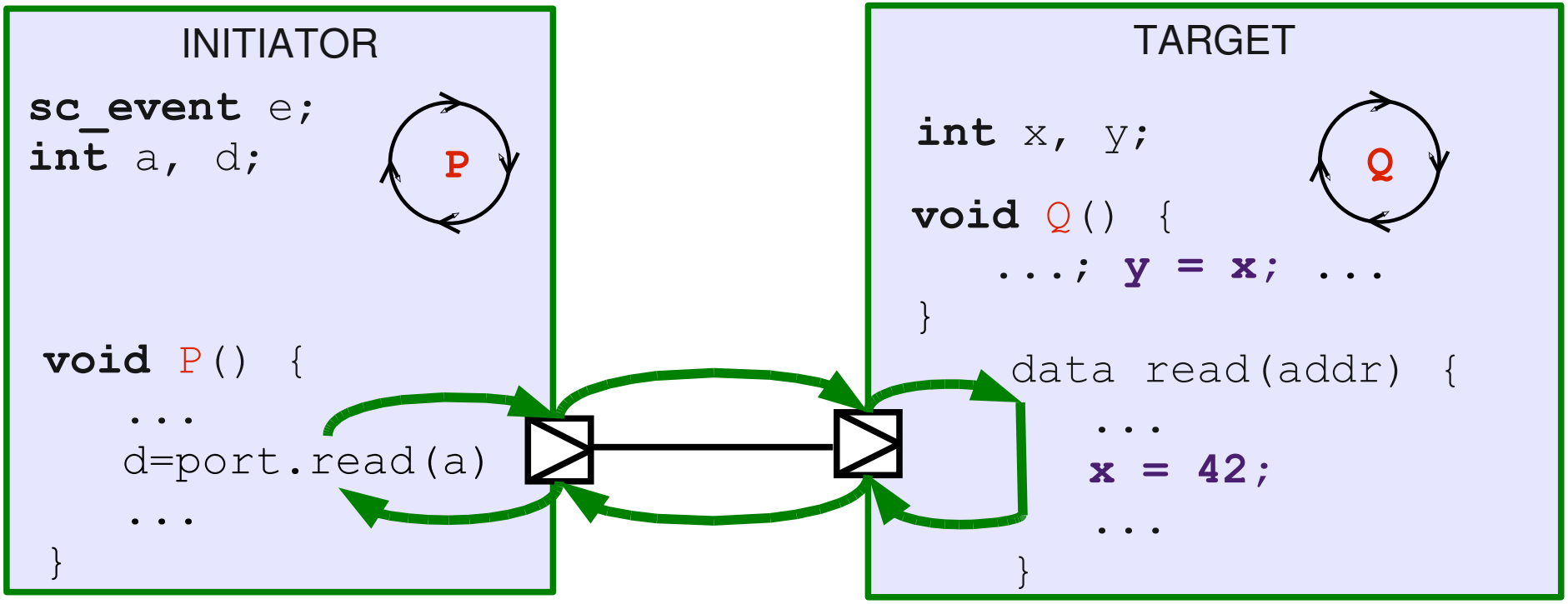
Based on the architecture of the model (i.e., the SystemC modules)
Two tasks of distinct modules are independent

1) Philippes Combes (ST Genève PhD):
clusters
modify the language (new **sc_node_module** concept)
Static partition of the independent task done by the developer

2) Eric Paire (ST Grenoble):
Multicore machines (SMP)
SystemC models with **synchronous channels**
A prototype working on TLM models, but
introduce **new preemption points**
Do not preserve the **semantics**



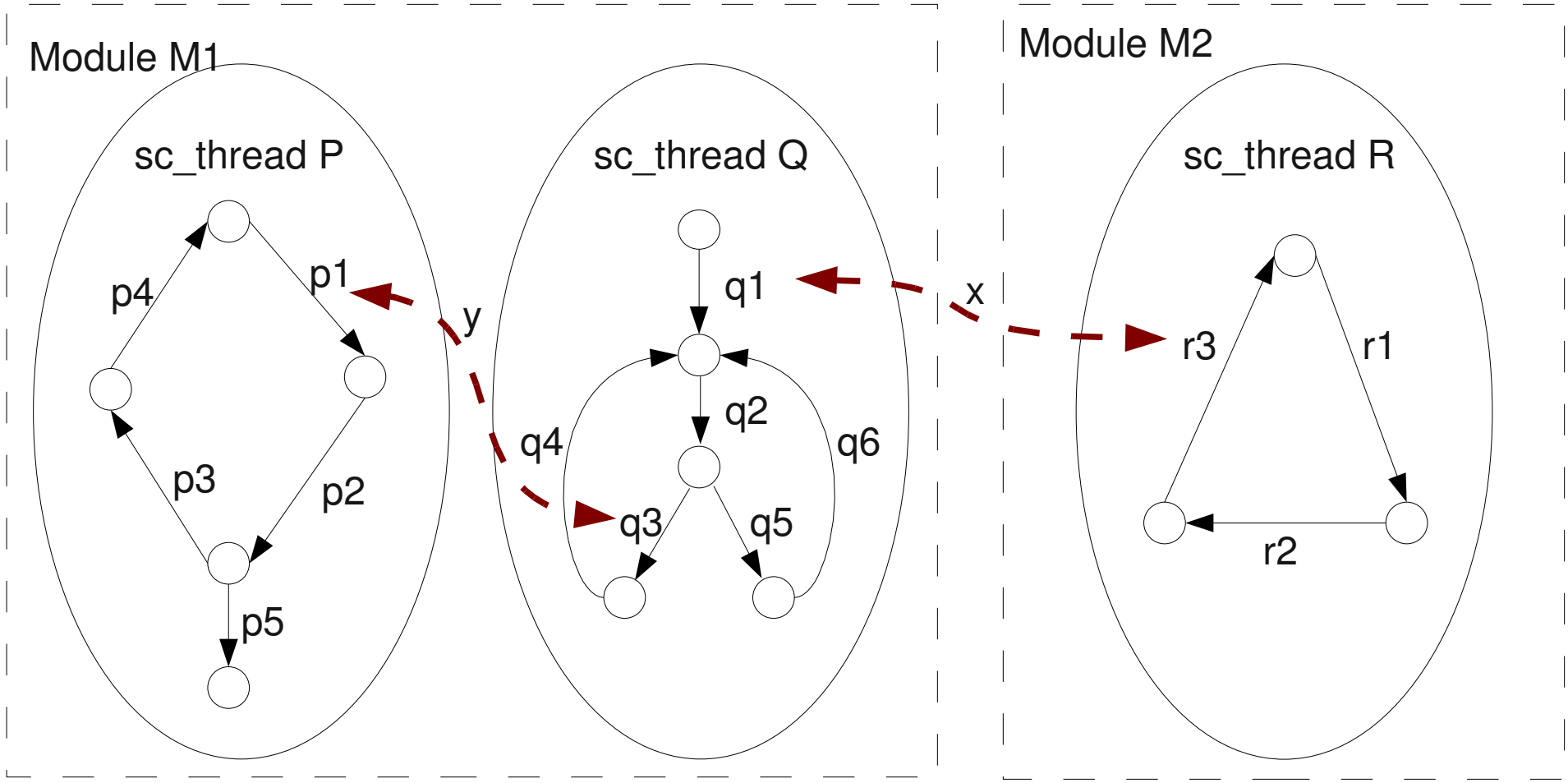
Structural independence and transactions



The transactions allow the processes to access data hosted by other modules.

Thus, transactions break the structural independency rule.

Many possible level for dependency analysis



If we look closer, we can parallelize more...

...but the analysis is harder

Our solution

Semantics-preserving parallelization requires dependency analysis
The best observation level for dependency analysis is the transition

Runtime algorithm
New scheduler for parallel simulation
Use both the Pthread (POSIX) librry and the QuickThread library
Protect shared memory internal to SystemC with mutexes and condition variables
Use a function indep(t1, t2) :
true iff t1 and t2 do not share any ressource

Combined with a static algorithm
Static dependency analysis to provide the indep(t1,t2) function
Identify the places where processes may yield back to the scheduler
Map each yield place to a list of maybe-accessed ressources

Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

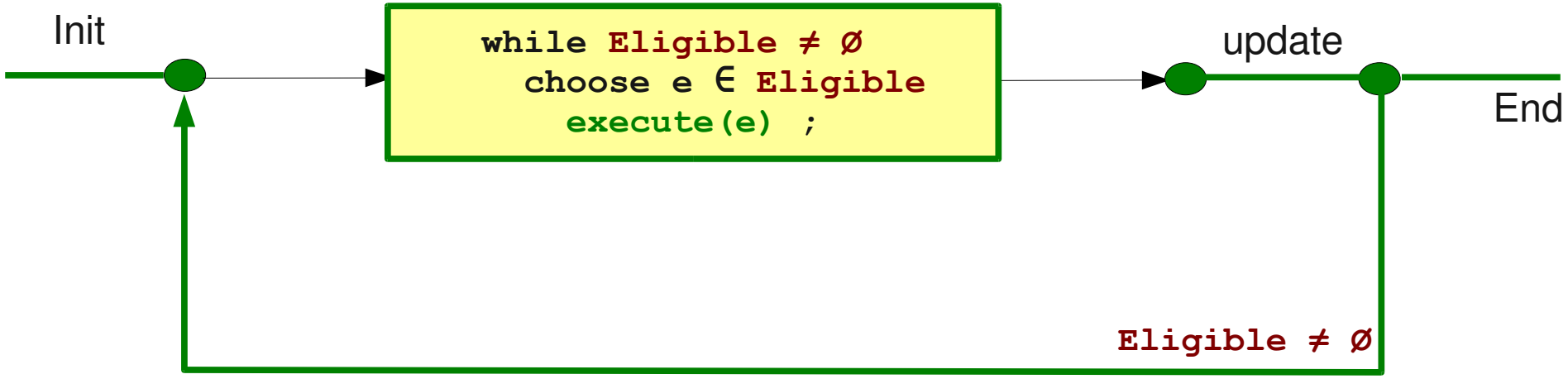
Algorithm and implementation

Experiments and results

Conclusion and further work

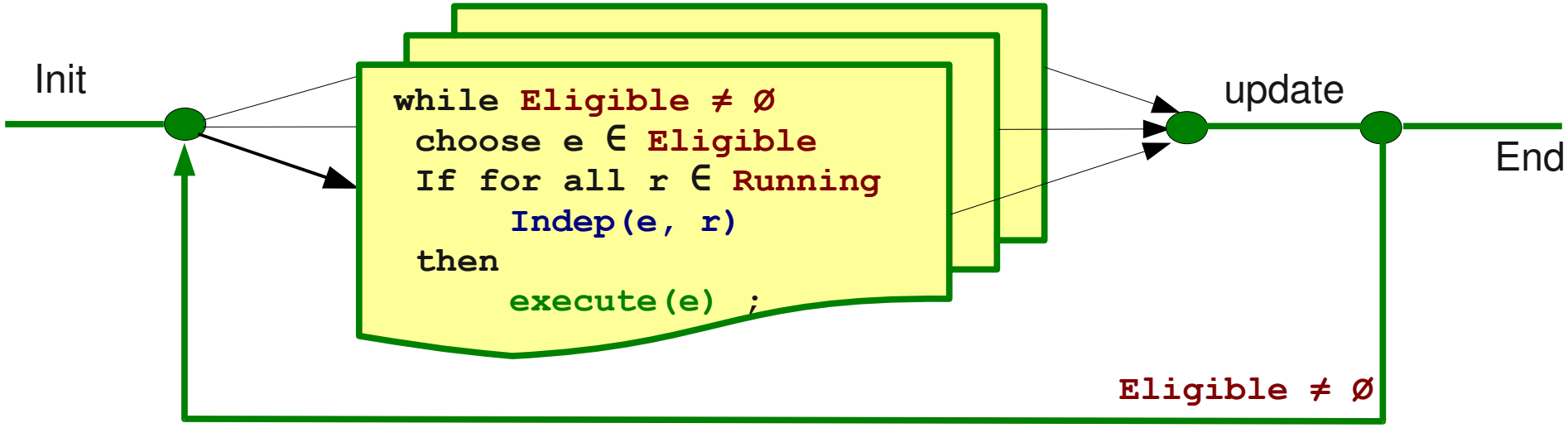
Sequential scheduler

Eligible : set of all eligible processes



Parallel scheduler

One « scheduler » per PThread
Each « scheduler » executes the same algorithm
`Eligible` : set of all the eligible processes
`Running` : set of all the processes being executed



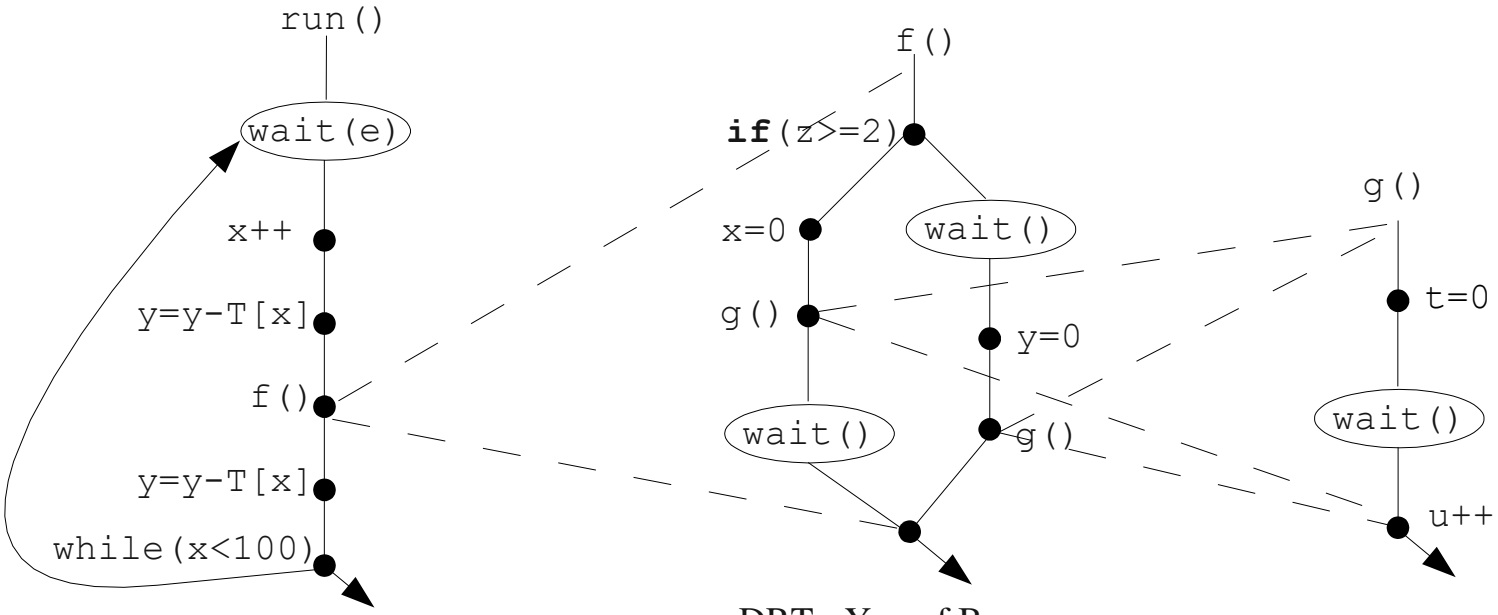
`Eligible` and **`Running`** protected by a mutex
Mutex released during **`execute(e)`**
Another tool must provide the function **`Indep(e, r)`**

Static analysis

```
sc_event e;
int x, y, z, t, u;
int * T;
void run(){
  do {
     wait(e);
     x++;
     y = y-T[x];
     f();
     y = T[0];
  } while(x<100);
}
```

```
void f(){
     if(z >= 2) {
          x = 0;
          g();
          wait();
     }
     else {
          wait();
          y = 0;
          g();
     }
}
```

```
void g(){
     t = 0;
     wait();
     u++;
}
```



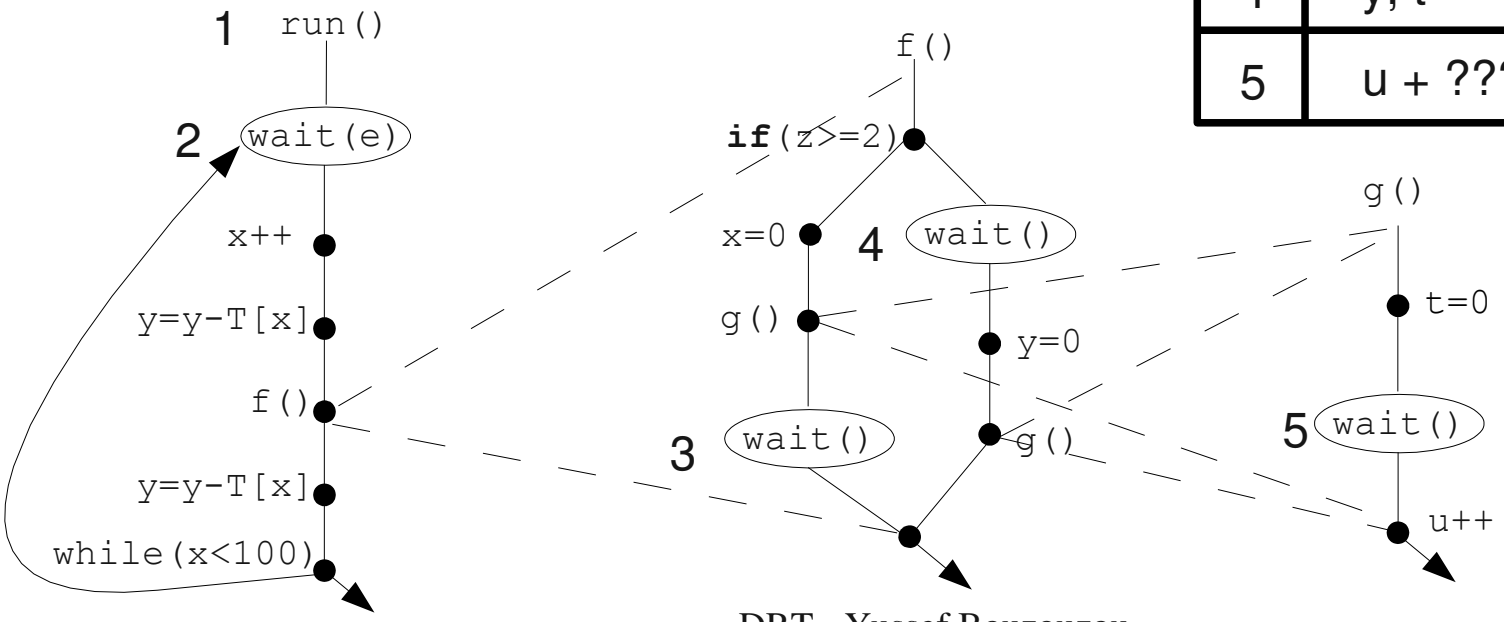
Static analysis

```
   sc_event e;
   int x, y, z, t, u;
   int * T;
1  void run(){
     do {
2      wait(e);
       x++;
       y = y-T[x];
       f();
       y = T[0];
     } while(x<100);
   }
```

```
void f(){
    if(z >= 2) {
        x = 0;
        g();
3       wait();
    }
    else {
4       wait();
        y = 0;
        g();
    }
}
```

```
void g(){
    t = 0;
5   wait();
    u++;
}
```

		return?
1	e	no
2	x, y, T, z, t	no
3	????	yes
4	y, t	no
5	u + ????	yes



Using the static analysis data during runtime

1	e	no
2	x, y, T, z, t	no
3	Ø	yes
4	y, t	no
5	u	yes

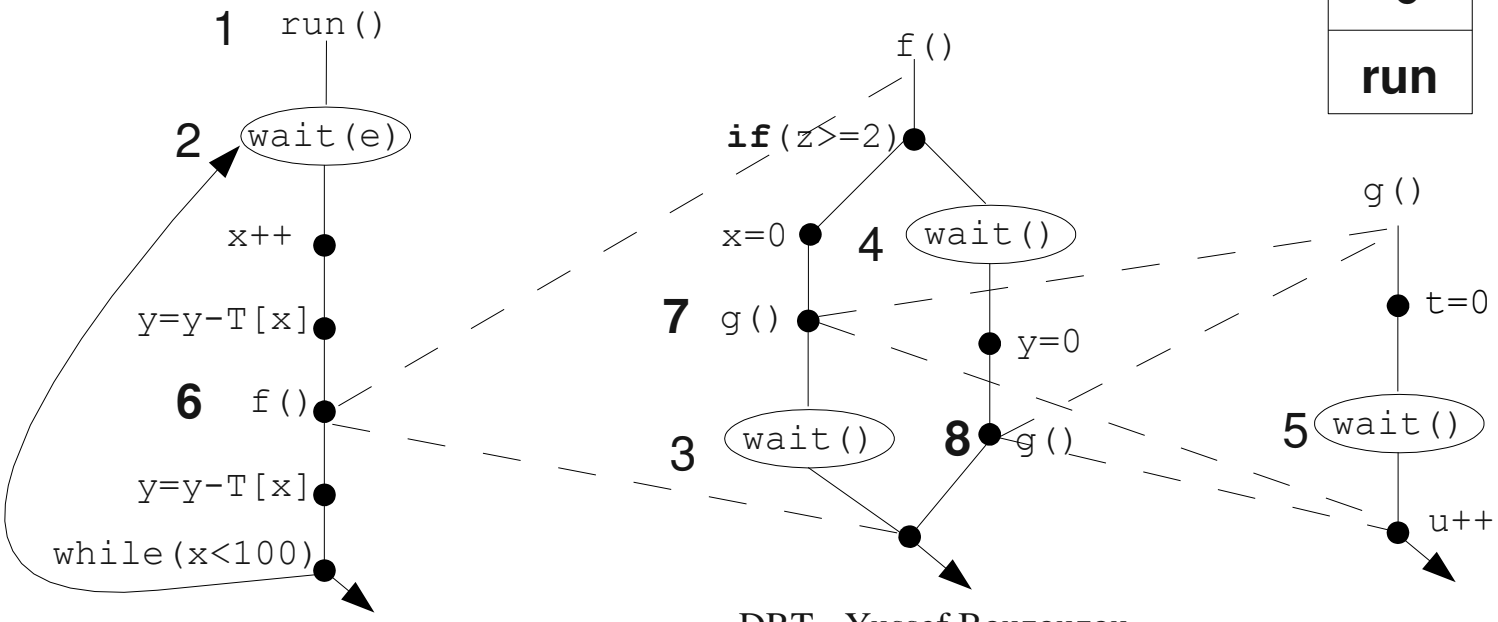
6	x, y, T, e	yes
7	Ø	no
8	Ø	yes

At runtime:

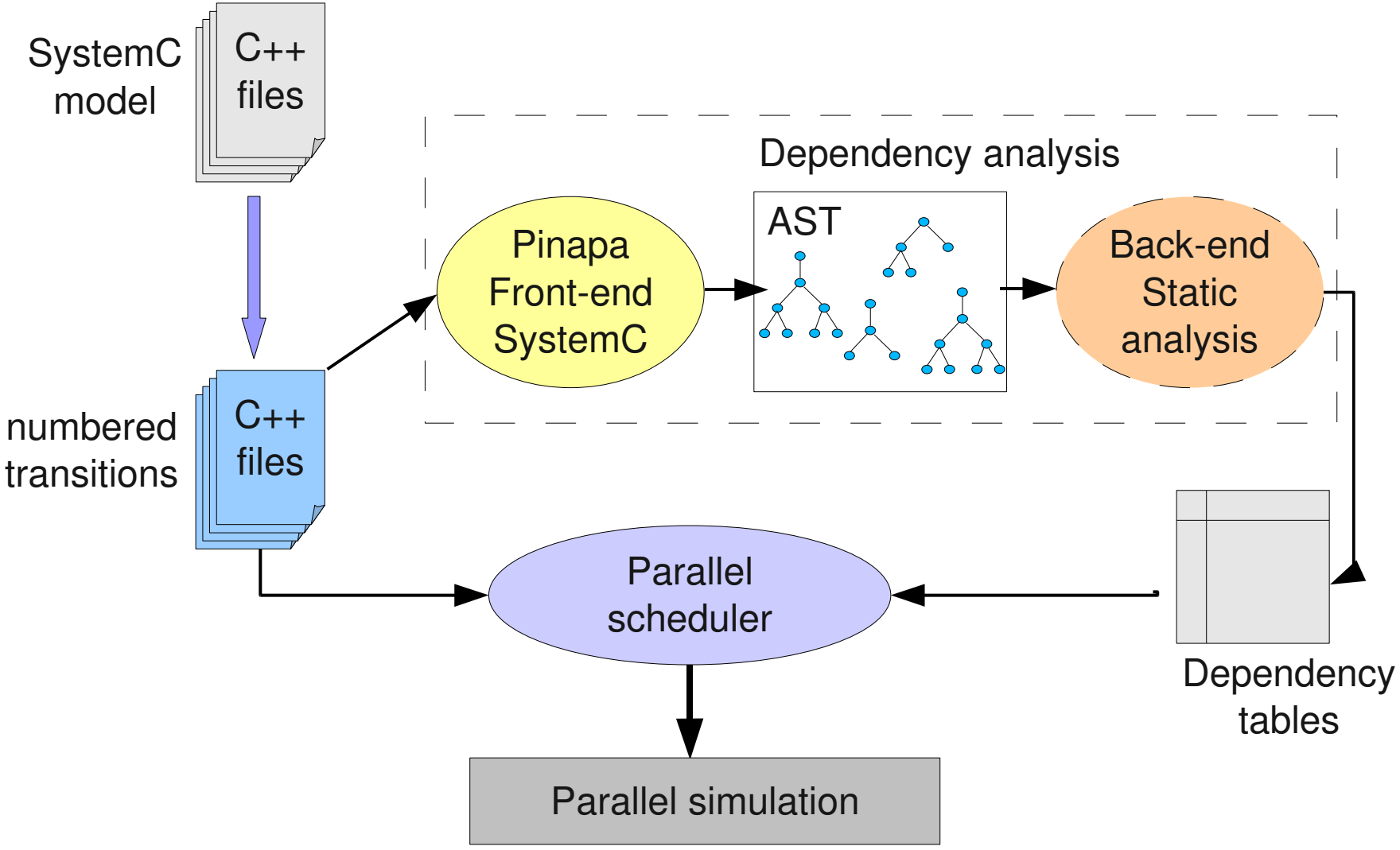
Run: 5 **+**

8	∪	Ø
6	∪	{ x, y, T, e }
run	∪	Ø

{ u }



Architecture of the parallel kernel



Outline

Context: simulation of systems-on-chip at the transaction level (TLM)

Parallel execution preserving the semantics

Algorithm and implementation

Experiments and results

Conclusion and further work

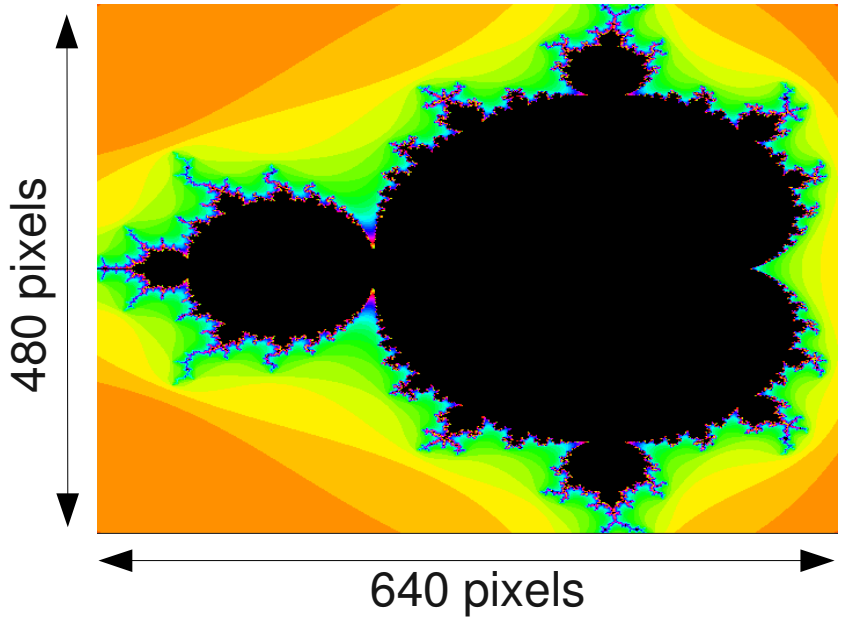
Experiments

Examples provided with the OSCI SystemC library
Not realistic enough, no at the TLM level
Allow to validate the parallel simulator

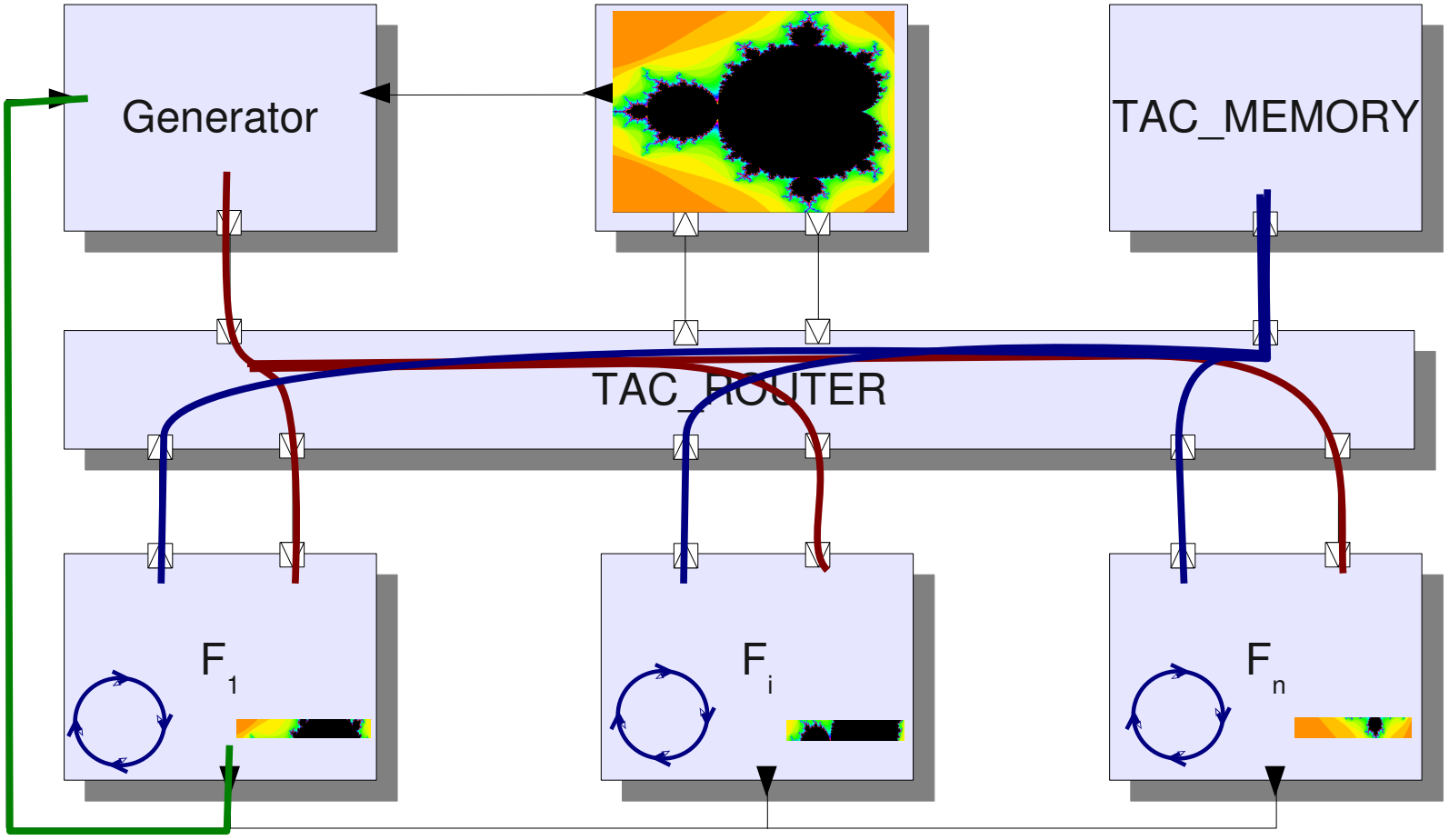
Examples provided by STMicroelectronics
Intrinsically sequential...
...or too much complicated to by analyzed by hand

Our case study: « Mandelbrot fractal »

- Mathematic function
- Result can be displayed as an image
- Each pixel can be computed independently
- Parallel computation of distinct parts
- Possible to parametrize the number of parts



Case study:
« Mandelbrot fractal»



Results

Addresses of the transaction: **port.read(address);**
It is impossible to compute statically a transaction address in the general case
A naive over-approximation is too strong:
Any target is possibly accessed (⇒ **no parallelization**)

The « RAM memory » (**Tac_memory**) :
All memory accesses are considered as **dependent**, because the RAM memory is considered as a whole
But in general each initiator accesses a different part of the memory

Loose timing annotations with randomization (**pv_wait(50,SC_SEC)**) .

Results : 4 processors, 5 Pthreads
4, 8, 16, 160 instances of the « Fractal » component
The parallel simulator is about 3 times as fast as the sequential one.

conclusion and further work

We proposed an algorithm for a **semantics-preserving SystemC parallel**

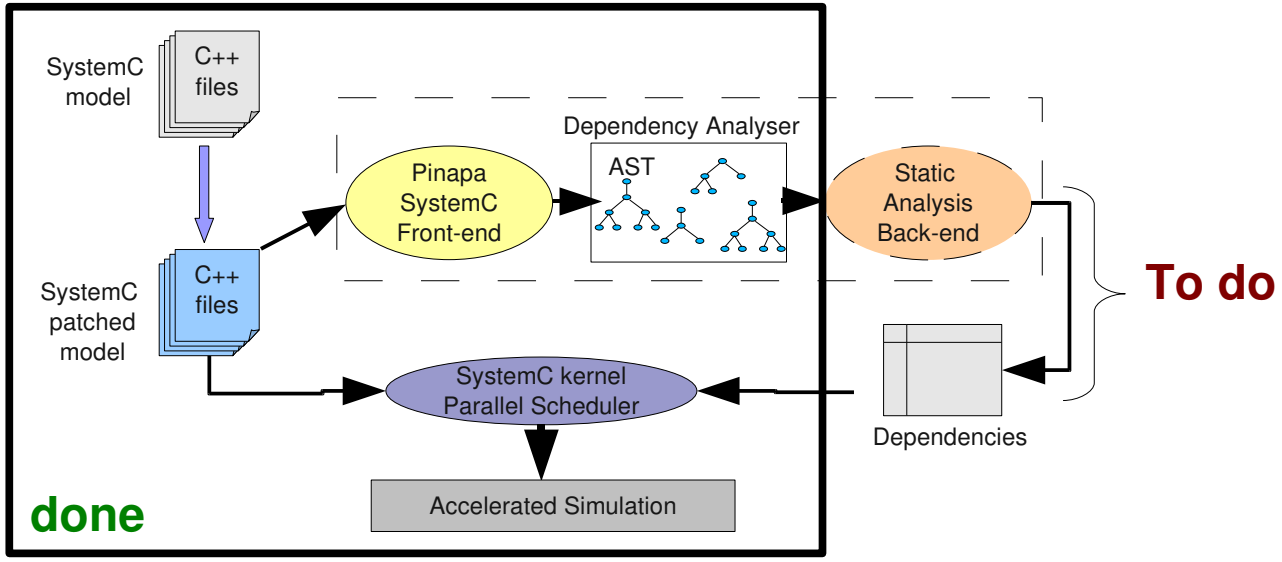
Implementation of a **parallel SystemC simulator**

Fully working on one class on SystemC models

(synchronous channels, no transactions)

Static analysis algorithms and **architecture** of a parallel SystemC framework

Idenfication of the efficiency issues



conclusion and further work

Decoding transactions addresses
Numeric abstract interpretation
Additional data provided by the user

Partition of the RAM memory: Idem

pv_wait (delay, unit) :
Select a « favorable » value

Modify the TLM library:
Non-atomic transaction (mp_read(), mp_write()...)
Only the bus need to be modified