Simulation in Metropolis

Guang Yang
10/28/2004

Outline

- Introduction
- SystemC-based Simulation
  - Implementing MMM Semantics
    - Imperative Constructs
    - Declarative Constraints
  - Efficient Simulation Techniques
- Case Study
- Conclusion

Introduction

- Platform based design
  - Platforms have sufficient flexibility to support a series of products
  - Choose a platform by design space exploration
  - Above two require models to be reusable
- Orthogonalization of concerns
  - Computation vs. Communication
  - Behavior vs. Coordination
  - Behavior vs. Architecture
  - Capability vs. Cost
- **Challenges**
  - **Relate orthogonalized concerns**
  - **Potential big overhead in design analysis**

Metropolis Meta-Model

- A combination of imperative program and declarative constraints
- Imperative program:
  - objects (process, media, quantity, statemedia)
  - netlist
  - await
  - block and label
  - interface function call
  - quantity annotation
- Declarative constraints:
  - Linear Temporal Logic (LTL)
  - (synch)
  - Logic of Constraints (LOC)
- **Challenges**
  - **Simulate constructs with rich semantics like await**
  - **Enforce declarative constraints in simulation**

SystemC-based Simulation

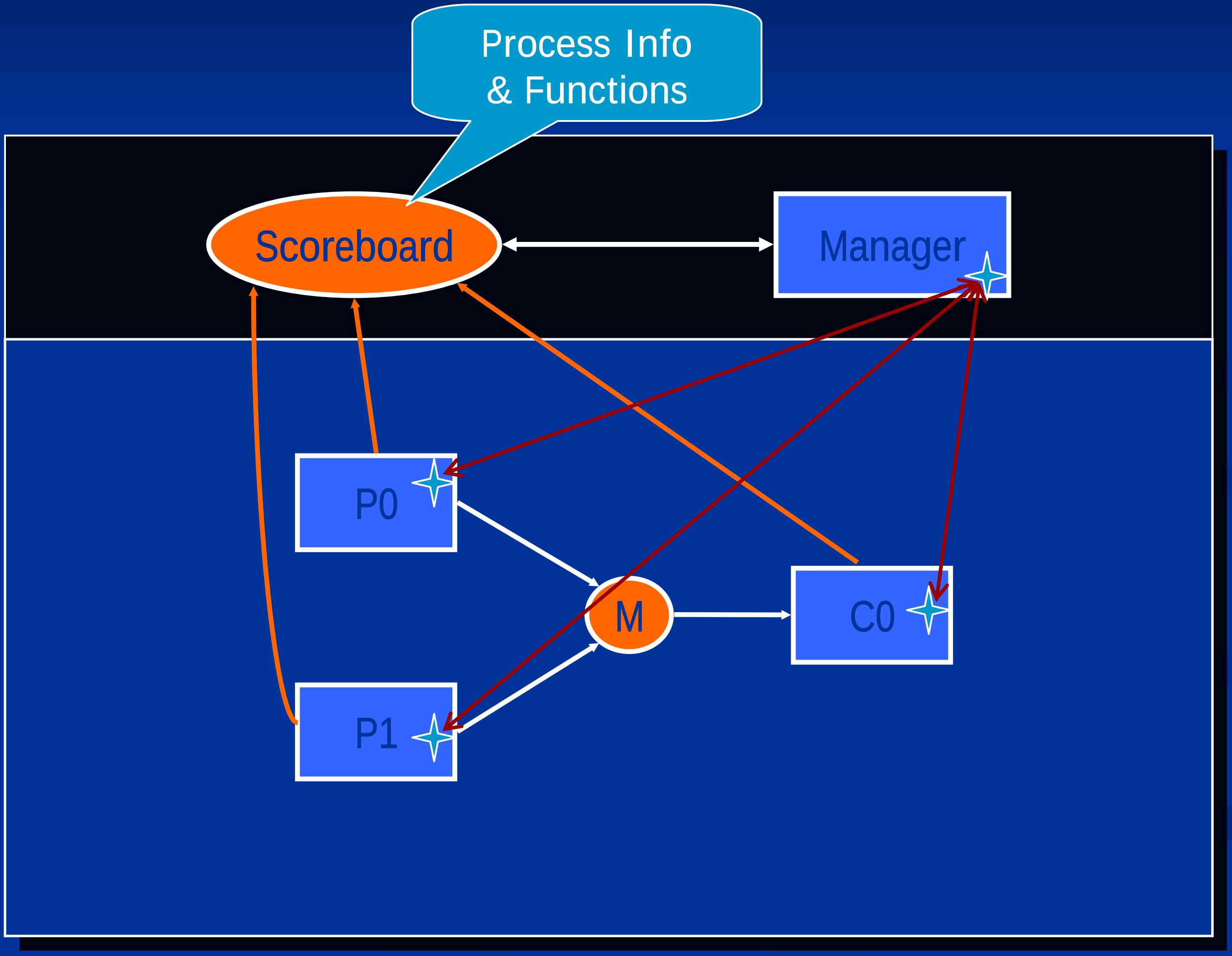
- Why SystemC Based Simulator?
  - Widely used by system designers
  - High simulation speed
  - Increasing number of supporting EDA tools
- Interleaving Concurrent Execution Semantics
  - sc_module
  - sc_channel
- Sequential Simulation Implementation
- http://www.systemc.org

- Challenges
  - Simulate constructs with rich semantics like await
  - Enforce declarative constraints in simulation

Implementing MMM Semantics

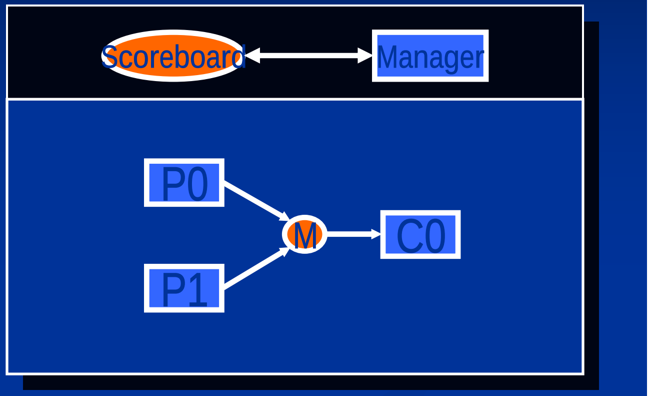
In MMM	In SystemC
process	sc_module
medium	sc_channel
statemedium	sc_channel
quantity	sc_channel
netlist	sc_channel, sc_main
port	sc_port
interface	sc_interface

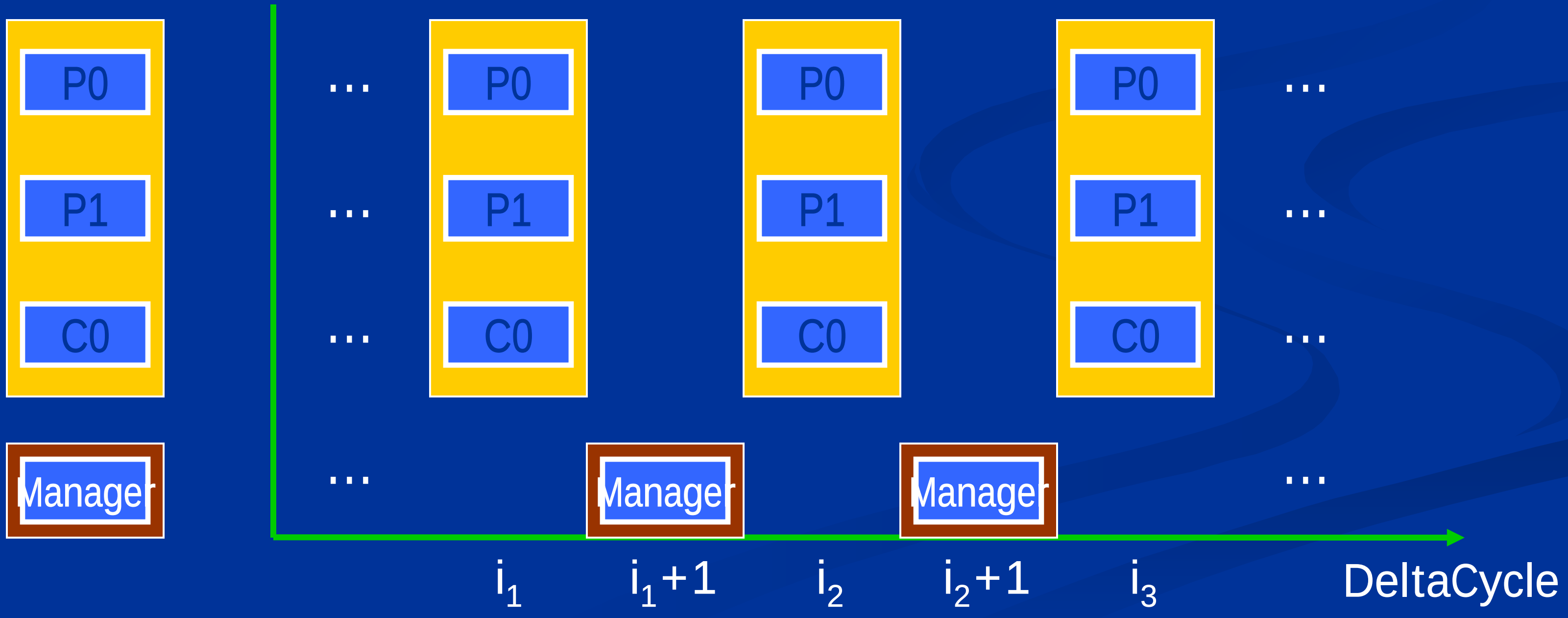
Overall Framework



Simulation Algorithm

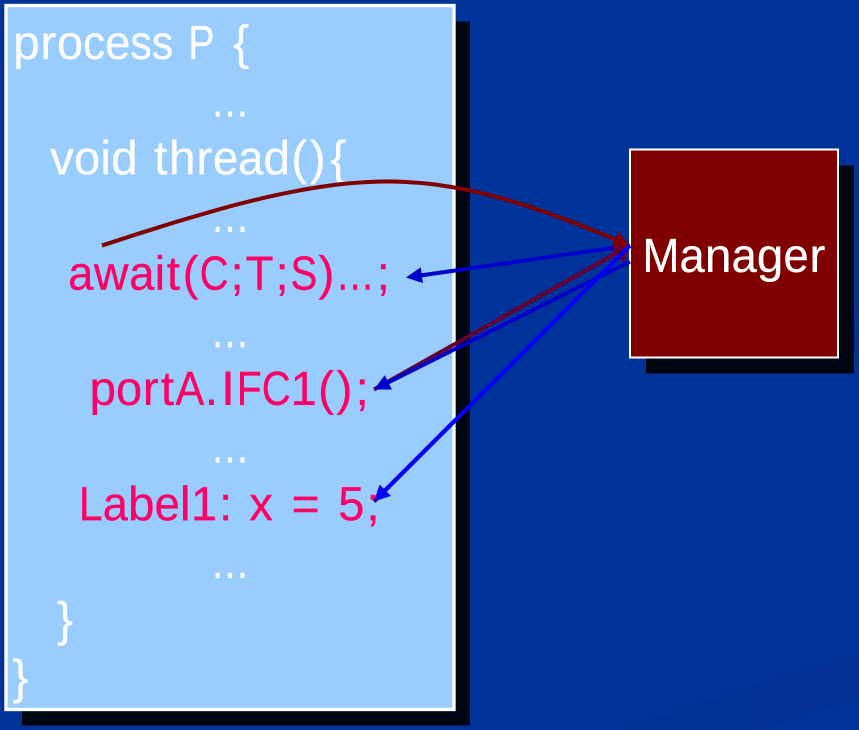
- Alternating running phases
  - Process phase
  - Manager phase





Simulation Algorithm (2)

- When to alternate?
  - At named events(currently await, IFC, label, block)
  - After manager makes decisions



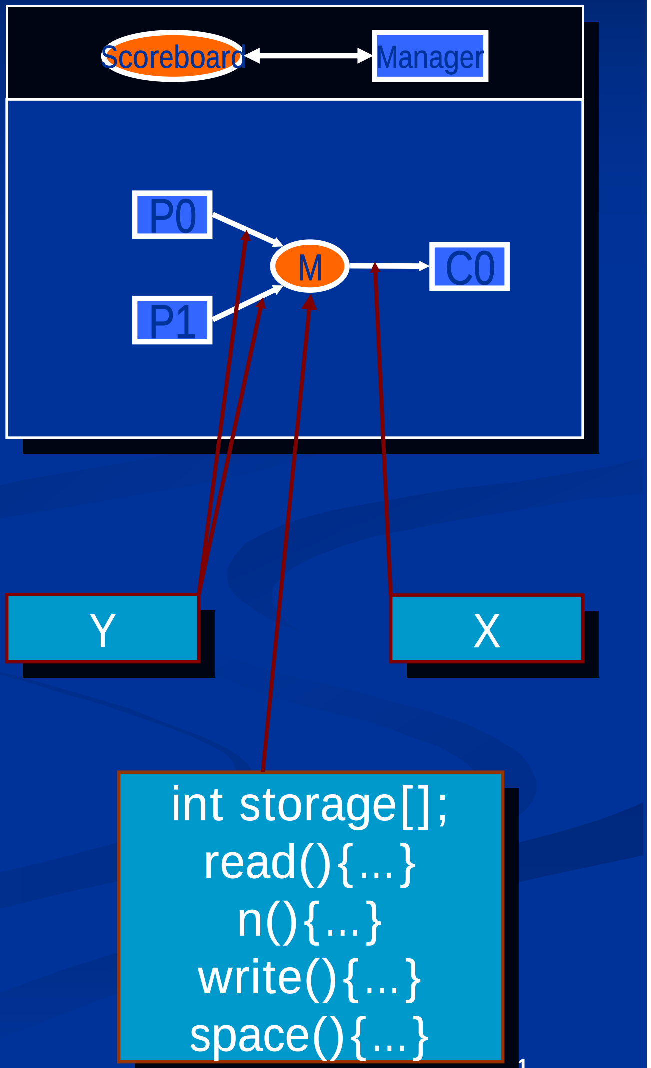
```
process P {
        ...
  void thread(){
        ...
    await(C;T;S)...;
        ...
    portA.IFC1();
        ...
    Label1: x = 5;
        ...
  }
}
```

Await Example

P0,P1,C0 prototype

```
process Proc {
  port reader X;
  port writer Y;

  void thread() {
    int w = 0;

    while (w < 30) {
      await {
        (Y.space() > 0; Y.writer; Y.writer) {
            Y.write(w);
            w = w + 1;    }
        (X.n() > 0; X.reader; X.reader) {
            X.read();    }
      }
    }
  }
}
```

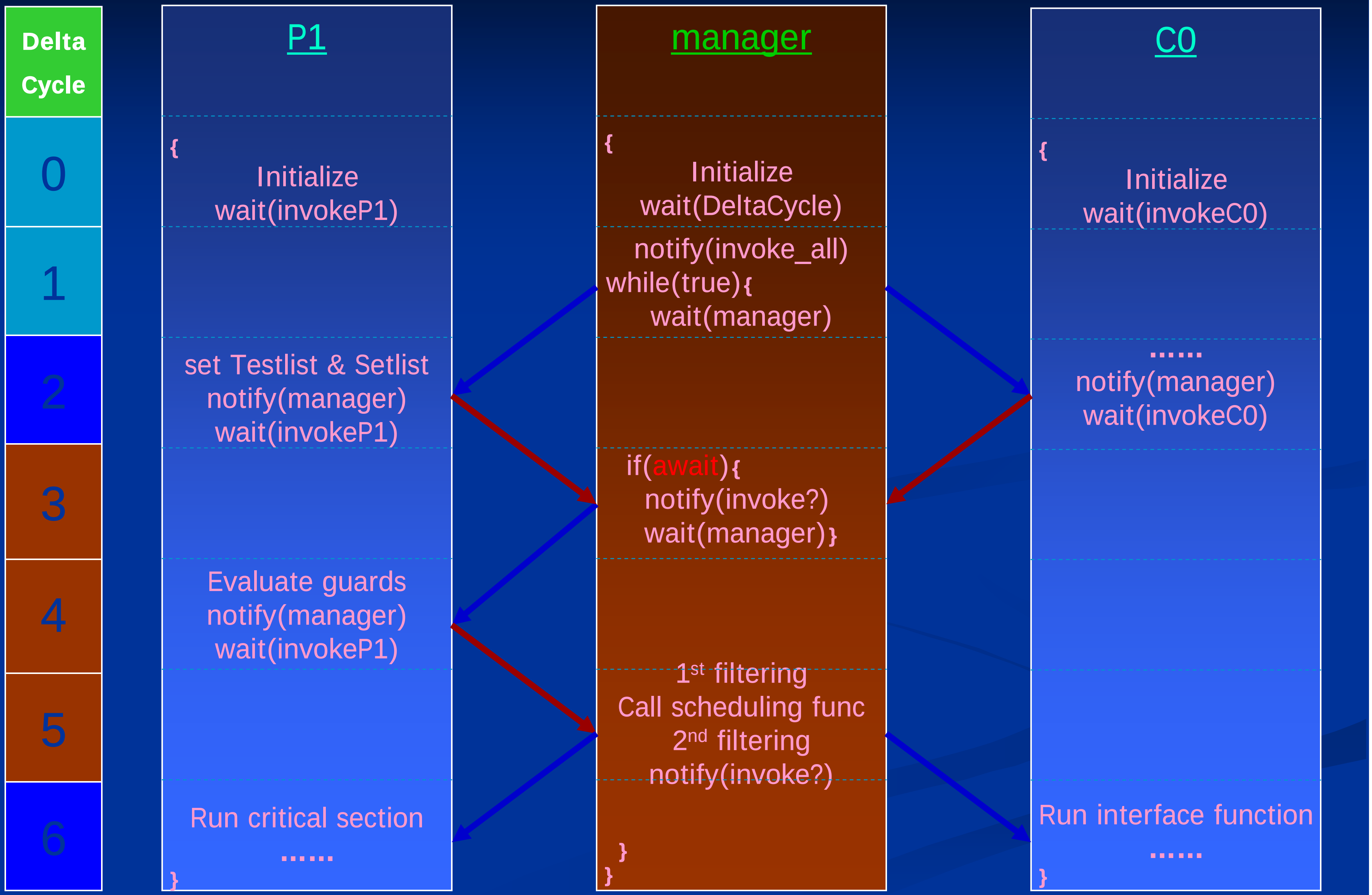
```
interface reader extends Port{
    update int read();
    eval int n();
}
interface writer extends Port{
    update void write(int i);
    eval int space();
}
```



int storage[];
read(){…}
n(){…}
write(){…}
space(){…}

```
await{(Y.space() > 0; Y.writer;  Y.writer)  Y.write(W);
      (       X.n() > 0; X.reader; X.reader) X.read(); }
```

X.read ();



Delta Cycle	P1	manager	C0

0	{
	Initialize
	wait(invokeP1)
		{
		Initialize
		wait(DeltaCycle)
			{
			Initialize
			wait(invokeC0)

1		notify(invoke_all)
		while(true){
		wait(manager)	

2	set Testlist & Setlist
	notify(manager)
	wait(invokeP1)
			......
			notify(manager)
			wait(invokeC0)

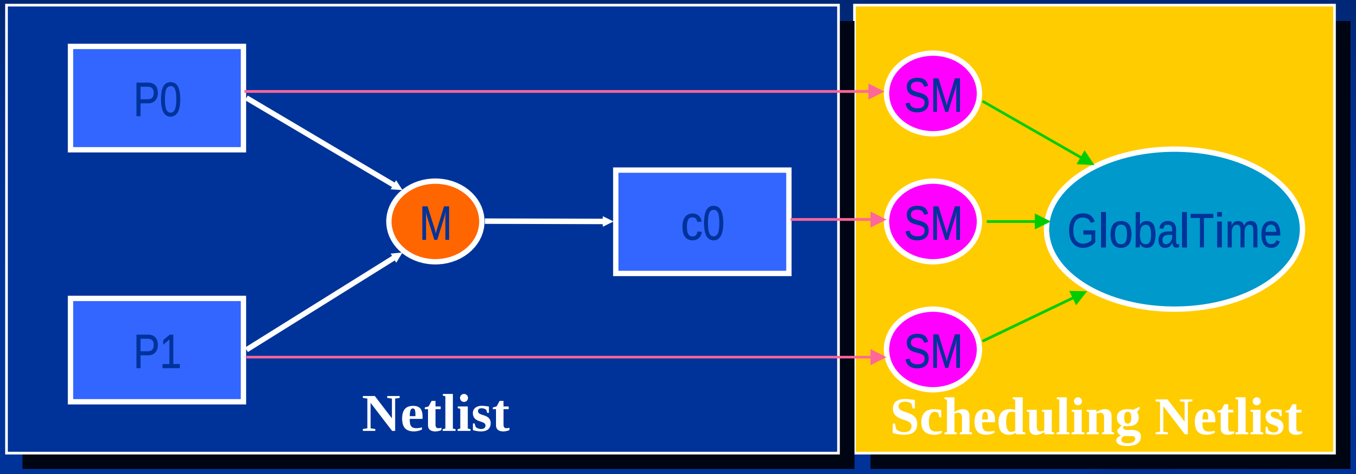
3		if(await){
		notify(invoke?)
		wait(manager)}	

4	Evaluate guards
	notify(manager)
	wait(invokeP1)		

5		1st filtering Call scheduling func 2nd filtering notify(invoke?)	
6	Run critical section ...... }	} }	Run interface function ...... }

Quantity

- System using GlobalTime quantity

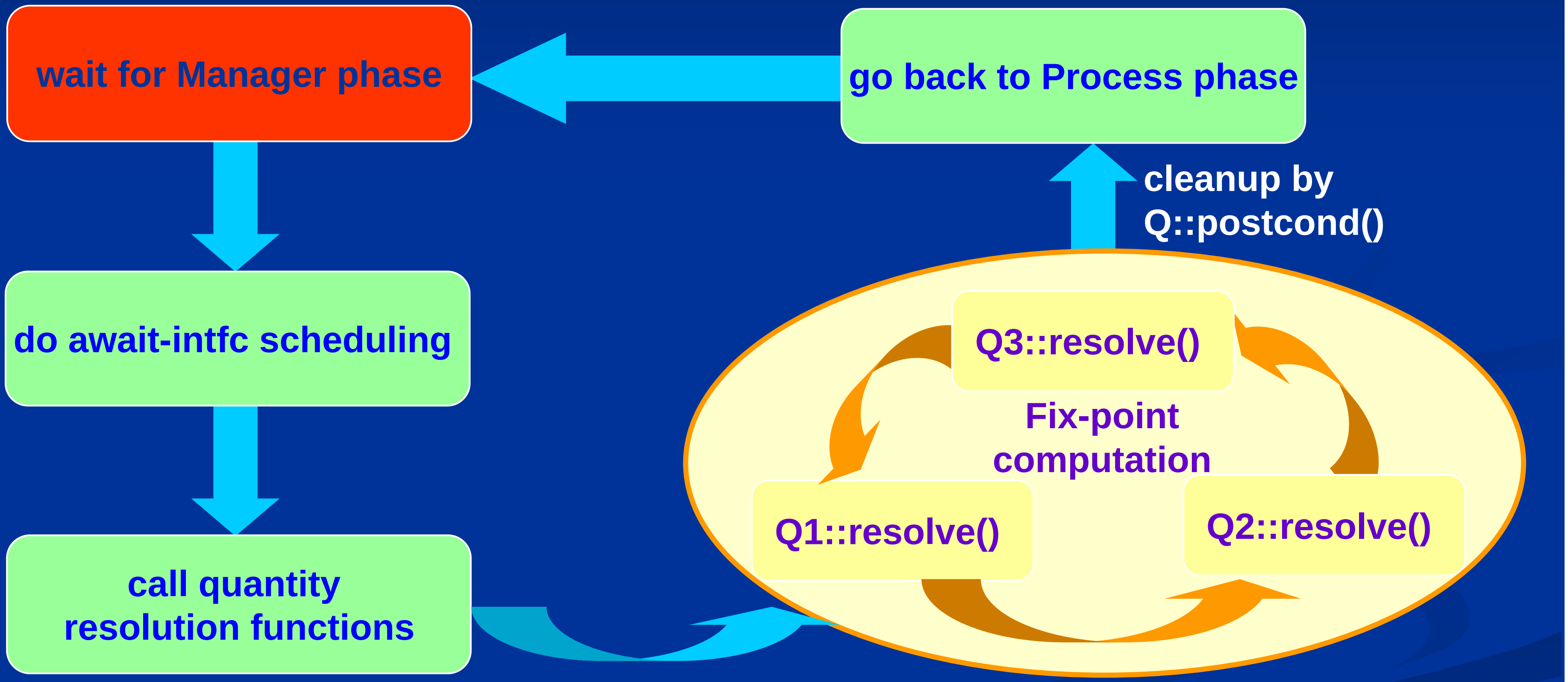


- Make request

```
process P{
  port writer Y;
  thread(){
   while(true){
    z=z+1;
   Y.write(z);
}}}
```

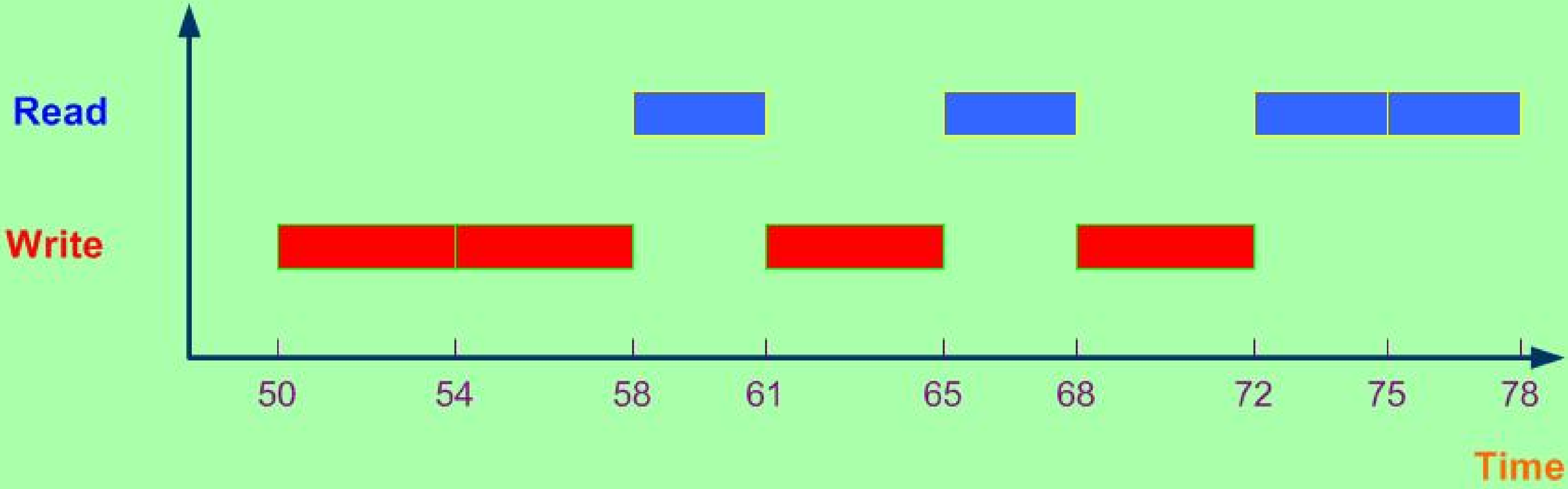
```
beg{    port2SM.requestI( beg_event  , 0);  }

end{    begTime=port2SM.A(beg( beg_event, LAST);
        port2SM.request(end( end_event, begTime+4));  }
```

Quantity Resolution in Simulation



Simulation Result

A sample of simulation result



- Challenges
  - Simulate constructs with rich semantics like await
  - Enforce declarative constraints in simulation

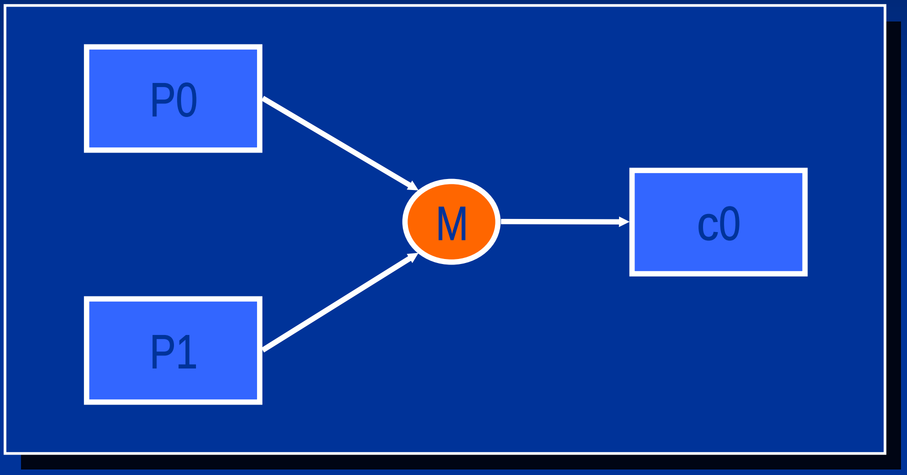
Constraints

- Logic of constraints (LOC)
  - Specify quantitative properties
    - e.g. throughput, rate, latency
  - can be checked by simulation+LOC checker
  - can be enforced by simulator
  - can be formally verified
- Linear Temporal Logic (LTL)
  - Defined over events, variables, etc.
  - Standard temporal operators, boolean operators

Enforcing LTL Constraints

- Basic idea
  - Convert LTL to Büchi Automaton (BA)
  - Keep track of system state and BA
  - Use BA to guide simulation

LTL constraints

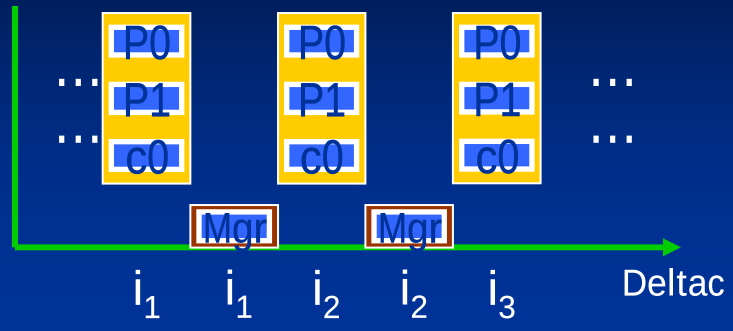


```
process P{
  port writer Y;
  thread(){
   while(true){
    z=z+1;
   Y.write(z);
}}}
```

```
constraint{
    // mutual exclusion between P0 and P1
    ltl( G( beg(P0, M.write) -> ( (! beg(P1, M.write)) U end(P0, M.write) ) ) );
    ltl( G( beg(P1, M.write) -> ( (! beg(P0, M.write)) U end(P1, M.write) ) ) );
}  } }
```

LTL constraints in Simulation

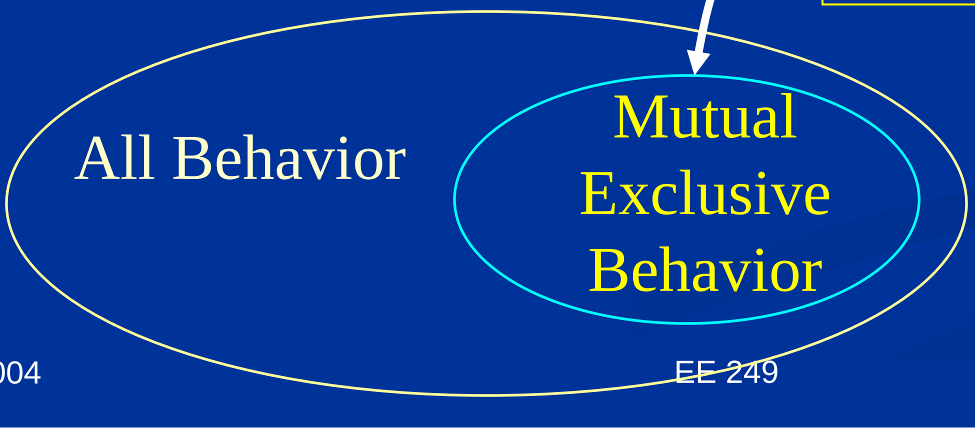
Process Phase



Manager Phase

```
switch to Manager phase
wait
Y.write(z);
switch to Manager phase
wait
```

```
Build Büchi Automaton for LTL
loop{
   wait
   do await-intfc scheduling
   choose good transition in BA
   switch to Process phase
}
```



Simulation Result

violation

Without LTL Enforcement

```
monitor> c     read      BEGIN
monitor> P0   write      BEGIN
monitor> P1   write      BEGIN
monitor> c     read      END
monitor> P0   write      END
monitor> P1   write      END
monitor> c     read      BEGIN
monitor> P0   write      BEGIN
monitor> P1   write      BEGIN
monitor> c     read      END
monitor> P0   write      END
monitor> P1   write      END
monitor> c     read      BEGIN
monitor> P0   write      BEGIN
monitor> P1   write      BEGIN
monitor> c     read      END
……
```

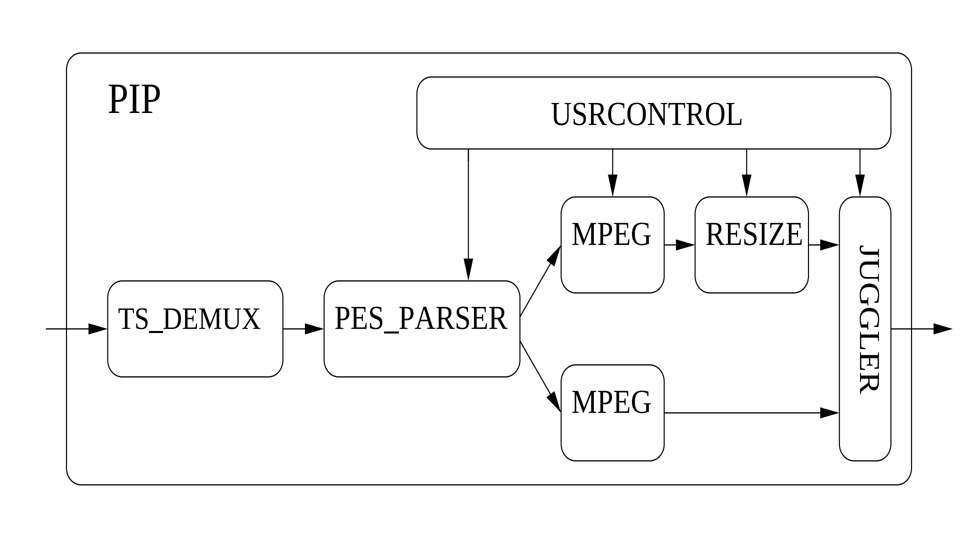
With LTL Enforcement

```
monitor> c    read       BEGIN
monitor> P0  write       BEGIN
monitor> c    read       END
monitor> P0  write       END
monitor> P1  write       BEGIN
monitor> c    read       BEGIN
monitor> P1  write       END
monitor> c    read       END
monitor> P0  write       BEGIN
monitor> c    read       BEGIN
monitor> P0  write       END
monitor> P1  write       BEGIN
monitor> c    read       END
monitor> P1  write       END
monitor> c    read       BEGIN
monitor> P0  write       BEGIN
……
```

NO violation

Case Study

- Picture-in-Picture



- 60 processes
- 200 media
- Approximately 19,000 lines of code

Advantages of Using Orthogonalization of Concerns

- Identified three critical errors in the behavior deadlocks: one in the algorithm and two in the communication protocols (first refinement step towards implementation)
- Quick architecture exploration
  - Changed rapidly different architectures
  - Changed rapidly communication mechanisms
- Analysis of interaction between algorithm choices and implementation architecture

Efficiency in Simulation

- Performance degradation w.r.t. native SystemC simulation (i.e., maintaining no separation of concerns)

Opt Tech	Sim. Time(s)	Cycle/ Second*
Baseline	7276	9.16K
Native SystemC	22.7	2.94M

Source: **Philips**
*: based on 200MHz clock

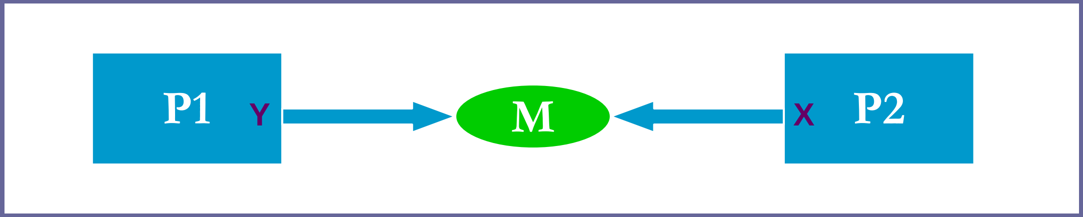
- Challenges
  - Relate orthogonalized concerns
  - Potential big overhead in design analysis

The Fix (Part 1): Optimization Techniques for Imperative Exclusion Constraints

- Medium-Centric Approach
  - Interface usage information is stored in media
  - Time complexity is linear in the number of processes
- Named Event Reduction
  - A named event is an event that needs to be observed → Record information and stop simulation at this event
  - Among the named events, static analysis could remove some unnecessary observance need
- Interleaving Concurrency

Exclusion constraints –
Interleaving Concurrency (1)

- Metropolis uses true concurrency
- The simulation platform, SystemC, uses interleaving concurrency



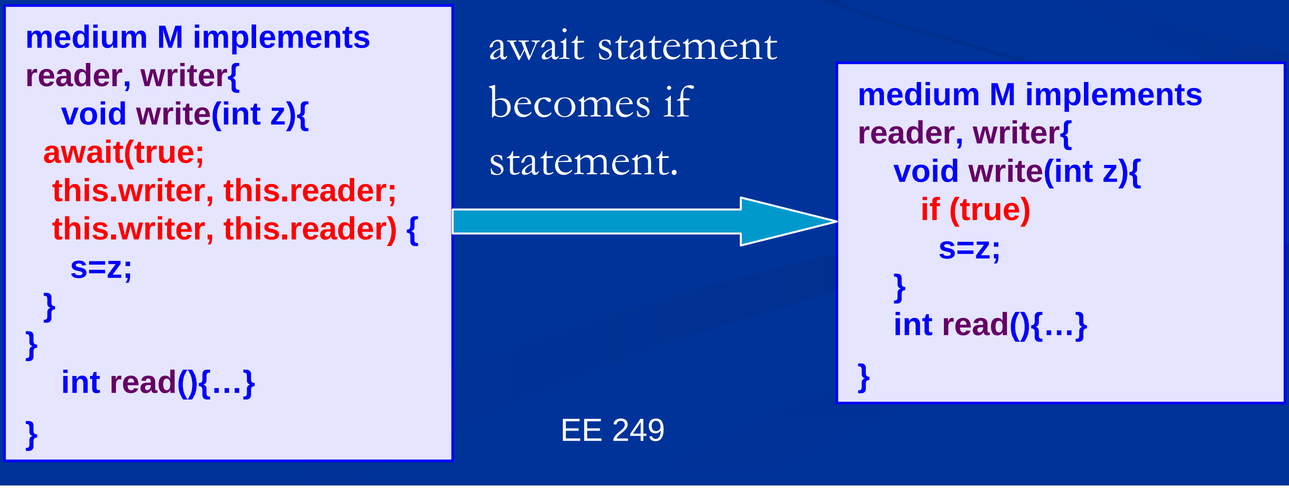
```
process P1{
  port writer Y;
  thread(){
   while(true){
    z=z*2;
    z=z+1;
    Y.write(z);
}}}
```

```
medium M implements
reader, writer{
    void write(int z){
  await(true;
   this.writer, this.reader;
   this.writer, this.reader) {
     s=z;
  }
}
    int read(){…}
}
```

```
process P2{
  port reader X;
  thread(){
   while(true){
    X.read();
}}}
```

Exclusion constraints –
Interleaving Concurrency (2)

- Interleaving implies that concurrent processes in Metropolis specification are scheduled on a sequential process
- Idea: take advantage of interleaving to make simulation faster



```
medium M implements
reader, writer{
    void write(int z){
  await(true;
   this.writer, this.reader;
   this.writer, this.reader) {
     s=z;
  }
}
    int read(){…}
}
```

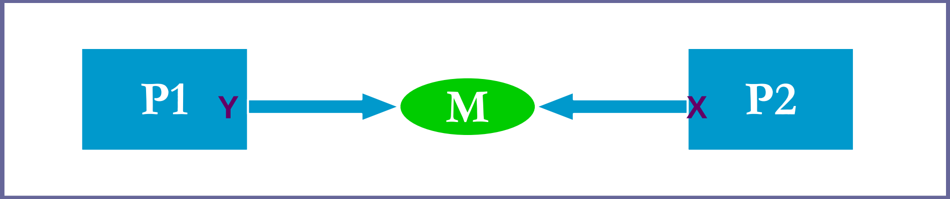
await statement becomes if statement.

```
medium M implements
reader, writer{
    void write(int z){
       if (true)
         s=z;
    }
    int read(){…}
}
```

Exclusion constraints –

Interleaving Concurrency (3)

- A sequence of events is Interleaving Concurrent Atomic (IC-Atomic) if no effective named events exists in that sequence of events.



```
process P1{
  port writer Y;
  thread(){
   while(true){
    z=z*2;
    z=z+1;
    Y.write(z);
}}}
```

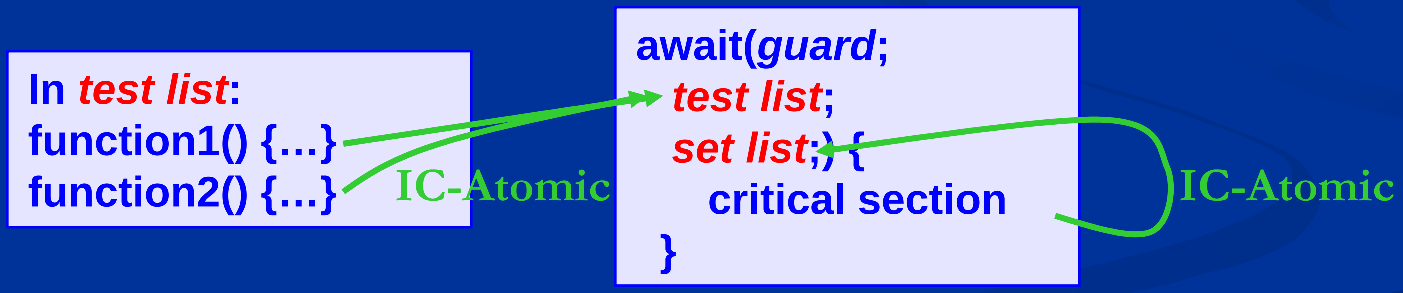
```
medium M implements
reader, writer{
    void write(int z){
  await(true;
   this.writer, this.reader;
   this.writer, this.reader) {
     s=z;
  }
}
    int read(){…}
}
```

```
process P2{
  port reader X;
  thread(){
   while(true){
    X.read();
}}}
```

Exclusion constraints –
Interleaving Concurrency (4)

- Theorem 1: For an await(*guard; test list; set list*) {*critical section*}, if *critical section* is IC-Atomic, and all interface functions in *test list* are IC-Atomic, then the await can be simplified to

  await(*guard*; ;) {*critical section*} or if (*guard*) {*critical section*}

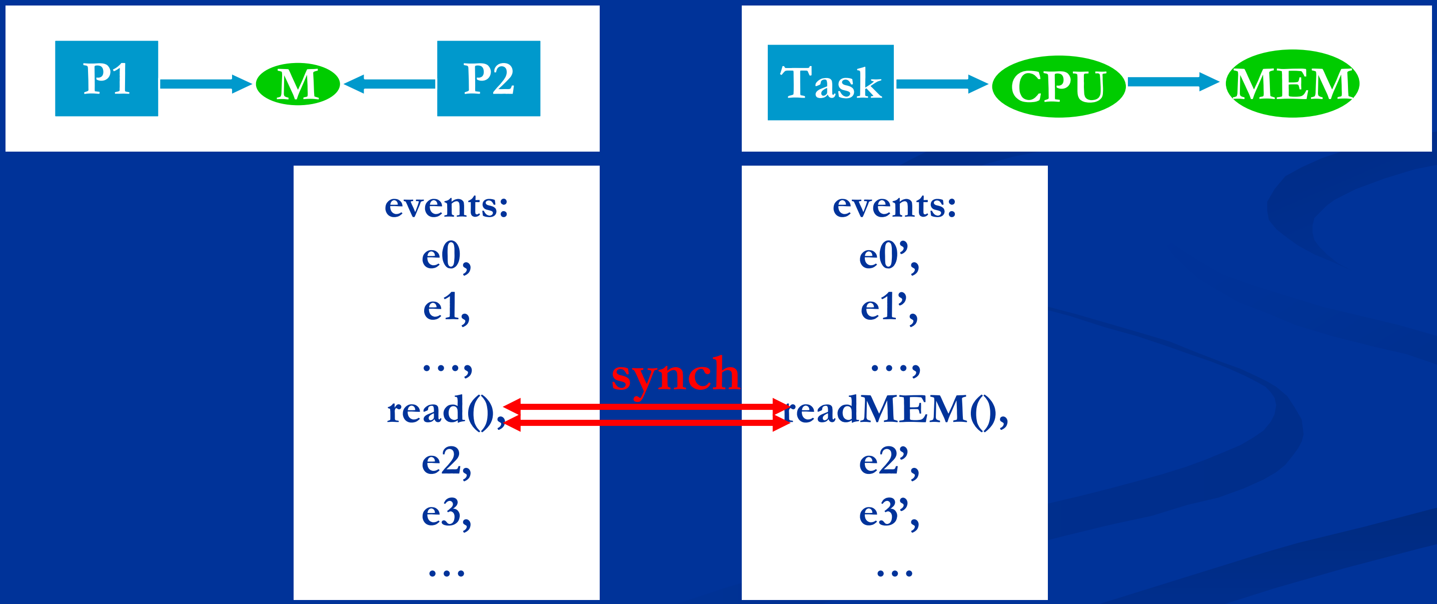


In ***test list***:
function1() {…}
function2() {…}

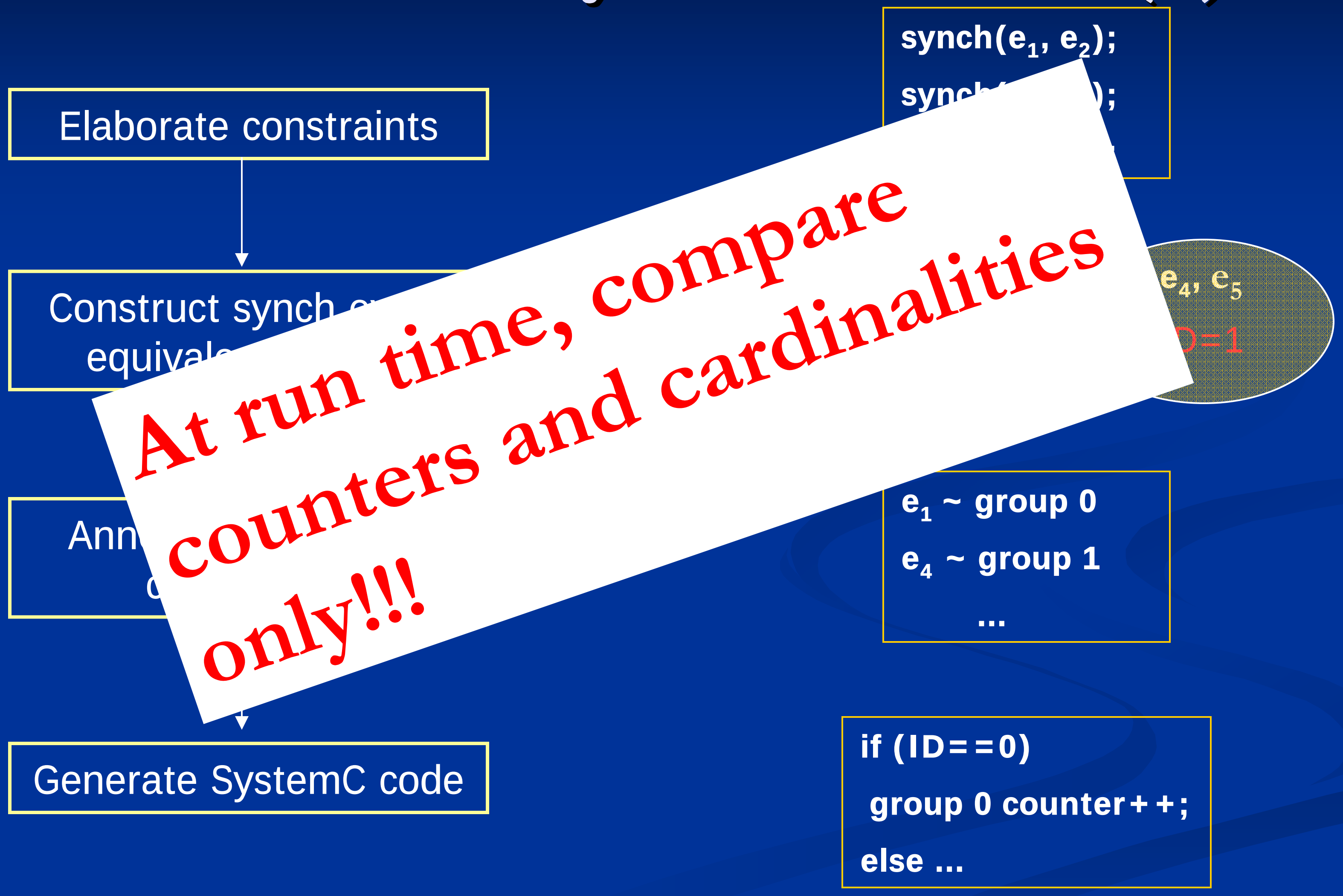
await(***guard***;
  test list;
  set list;) {
    critical section
  }

The Fix Part 2:
Constraints for coordinating sequential programs: Declarative Simultaneity Constraints

- *Declarative Simultaneity Constraints*: constraints separated from imperative programs
  - Can be used to specify behavior-architecture mapping



Simultaneity constraints(2)



synch(e_1, e_2);

synch( );

e_1 ~ group 0

e_4 ~ group 1

…

```
if (ID==0)
 group 0 counter++;
else ...
```

At run time, compare counters and cardinalities only!!!

Case Study (2)

- Picture-in-Picture behavior simulation result

Opt Tech	Sim. Time(s)	Cycle/ Second*	Overall Speedup	Speedup by
Baseline	7276	9.16K	1	---
MC	1797	37.1K	4	MC: 4
MC/NER	89.26	747K	80	NER: 20
MC/NER/IC	20.29	3.29M	359	IC: 4.5
Native SystemC	22.7	2.94M	---	---

- MC: Medium-Centric
- NER: Named Event Reduction
- IC: Interleaving Concurrency
- *: based on 200MHz clock

Case Study (3)

- PiP Behavior model + CPU-Bus-Mem model
- Behavior-Architecture Mapping

# of Simultaneity Constraints	Handling Overhead *
8	2.9%
16	2.9%
32	3.4%
64	4.0%

*: compared with the time spent on behavior and architecture themselves

Conclusion

- MMM language has strong expressive power. Simulation of the language is done on top of SystemC.
- Orthogonalizing concerns in system design is essential, but introduces overhead to analysis [illegible] general it could be huge.
- We applied a fe[illegible] the overhead. From th[illegible]ws 4X to 20X speedup of inc[illegible] Combine the techniques togeth[illegible]minate all overhead.

Efficient SystemC-based Metropolis Simulator!

Questions?