Validation of SoC models in presence of indeterministic schedulings and loose timings

Claude Helmstetter
with Florence Maraninchi, Laurent Maillet-Contoz, Matthieu Moy, Jérôme Cornet, ...

Verimag &
ST Microelectronics

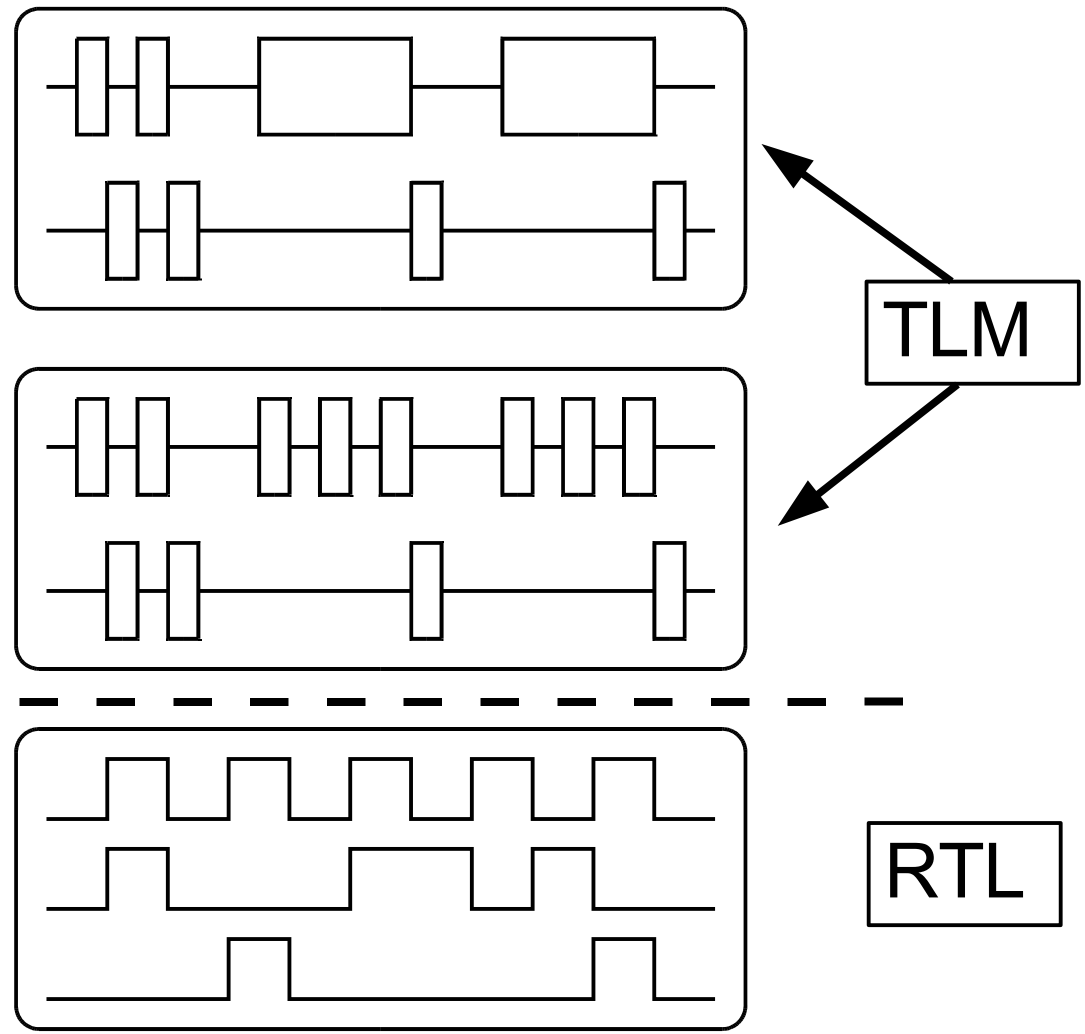
Outline

- Context: modeling of SoCs in SystemC-TLM
- Our Problem: managing scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

Context: Transaction Level Model

simulation speed

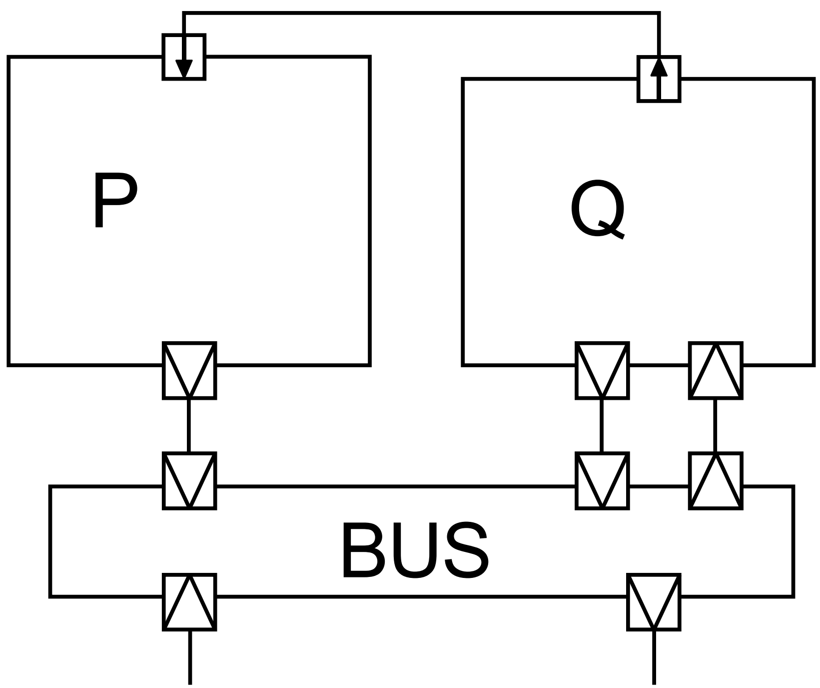
- Early simulation of the embedded software
- Golden model for RTL validation
- Architecture exploration



- SoC synthesis

accuracy

SystemC: C++ Library




```
...
unsigned x;
sc_event e;
SC_HAS_PROCESS(top);
top(sc_module_name
name):
      sc_module(name) {
  SC_THREAD(P);
  SC_THREAD(Q);
}
void top::P() {
  wait(e);
  ...
```

Construction of the architecture first, then non-preemptive scheduling, simulated time.

Examples

With fixed delays:

```
void top::P() {
    wait(e);
    wait(20);
    if (x) cout << "Ok\n";
    else cout << "Ko\n";}
```

||

```
void top::Q() {
    e.notify();
    x = 0;
    wait(20);
    x = 1;}
```

Examples

Untimed:

```
void top::P() {
    wait(e);
    wait(20);
    yield();
    if (x) cout << "Ok\n";
    else cout << "Ko\n";}
```

||

```
void top::Q() {
   e.notify();
   x = 0;
   wait(20);
   yield();
   x = 1;}
```

Examples

With loose delays:

```
void top::P() {
  lwait(3,d1); //t1
  wait(e);
  wait(20); yield();
  lwait(40,d2); //t2
  if (x) cout << "Ok\n";
  else cout << "Ko\n";}
```

||

```
void top::Q() {
  lwait(6,d3); //t3
  e.notify();
  x = 0;
  wait(20); yield();
  lwait(24,d4); //t4
  x = 1;}
```

Outline

- Context: modeling of SoCs in SystemC-TLM
- Our Problem: manage scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

Example of Scheduling Dependencies

```
      void top::P() {
P1       wait(e);
P2       wait(20);
P3       if (x) cout << "Ok\n";
         else cout << "Ko\n";}
```

||

```
void top::Q() {
   e.notify();        Q1
   x = 0;
   wait(20);
   x = 1;}            Q2
```

- **3** possible schedulings: (TE=Time Elapse)
  - P_1;Q_1;P_2;[TE];Q_2;P_3: **Ok**
    default OSCI scheduler choice, if **P** declared before **Q and** if ...
  - P_1;Q_1;P_2;[TE];P_3;Q_2: **Ko**
  - Q_1;P_1;[TE];Q_2: **“dead-lock”**

Example of Timing Dependencies

```
void top::P() {
   lwait(3,2); //t1
   wait(e);
   lwait(40,10); //t2
   if (x) cout << "Ok\n";
   else cout << "Ko\n";}
```

‖

```
void top::Q() {
   lwait(6,2); //t3
   e.notify();
   x = 0;
   lwait(24,6); //t4
   x = 1;}
```

- 3 possible executions again:
  - With $t_1 \rightarrow 3$, $t_2 \rightarrow 40$, $t_3 \rightarrow 6$, $t_4 \rightarrow 24$: **Ok**
  - With $t_1 \rightarrow \mathbf{5}$, $t_2 \rightarrow 40$, $t_3 \rightarrow \mathbf{4}$, $t_4 \rightarrow 24$: **dead-lock**
  - With $t_1 \rightarrow 3$, $t_2 \rightarrow \mathbf{30}$, $t_3 \rightarrow 6$, $t_4 \rightarrow \mathbf{30}$: **Ko** possible

The Coverage Problem

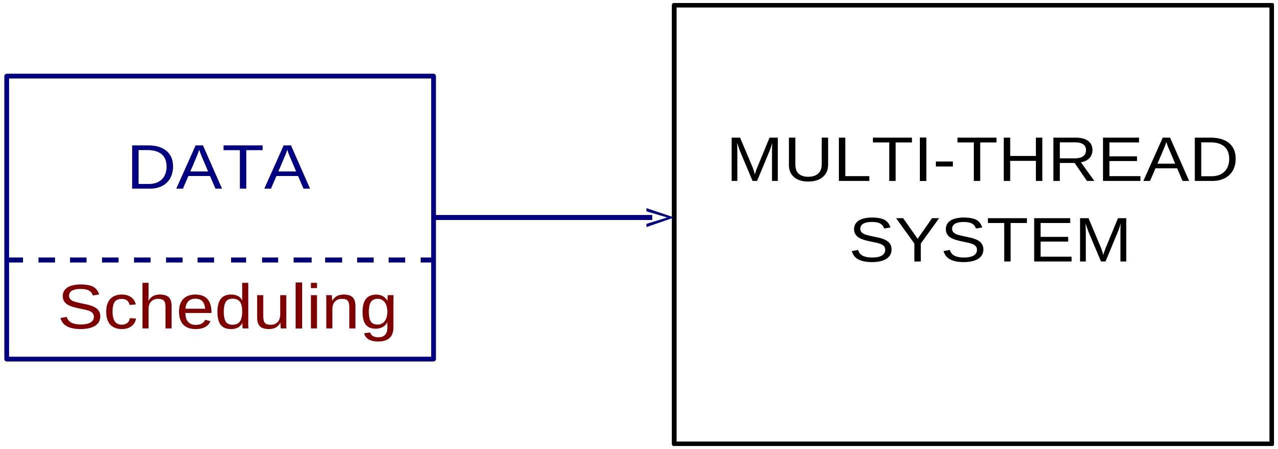
- Even if data is fixed
  - The SystemC LRM allows many schedulings
  - Delays may be not fixed (designer choice)
- For the validation of SoC models:
  - 1 execution $\Rightarrow$ very poor coverage
  - Random schedulings and timings => uncertain coverage, lots of useless executions
  - Test with all possible values => unrealistic
  - Our goal : **test only the executions that may lead to different final states**

Outline

- Context: modeling of SoCs in SystemC-TLM
- Our Problem: managing scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

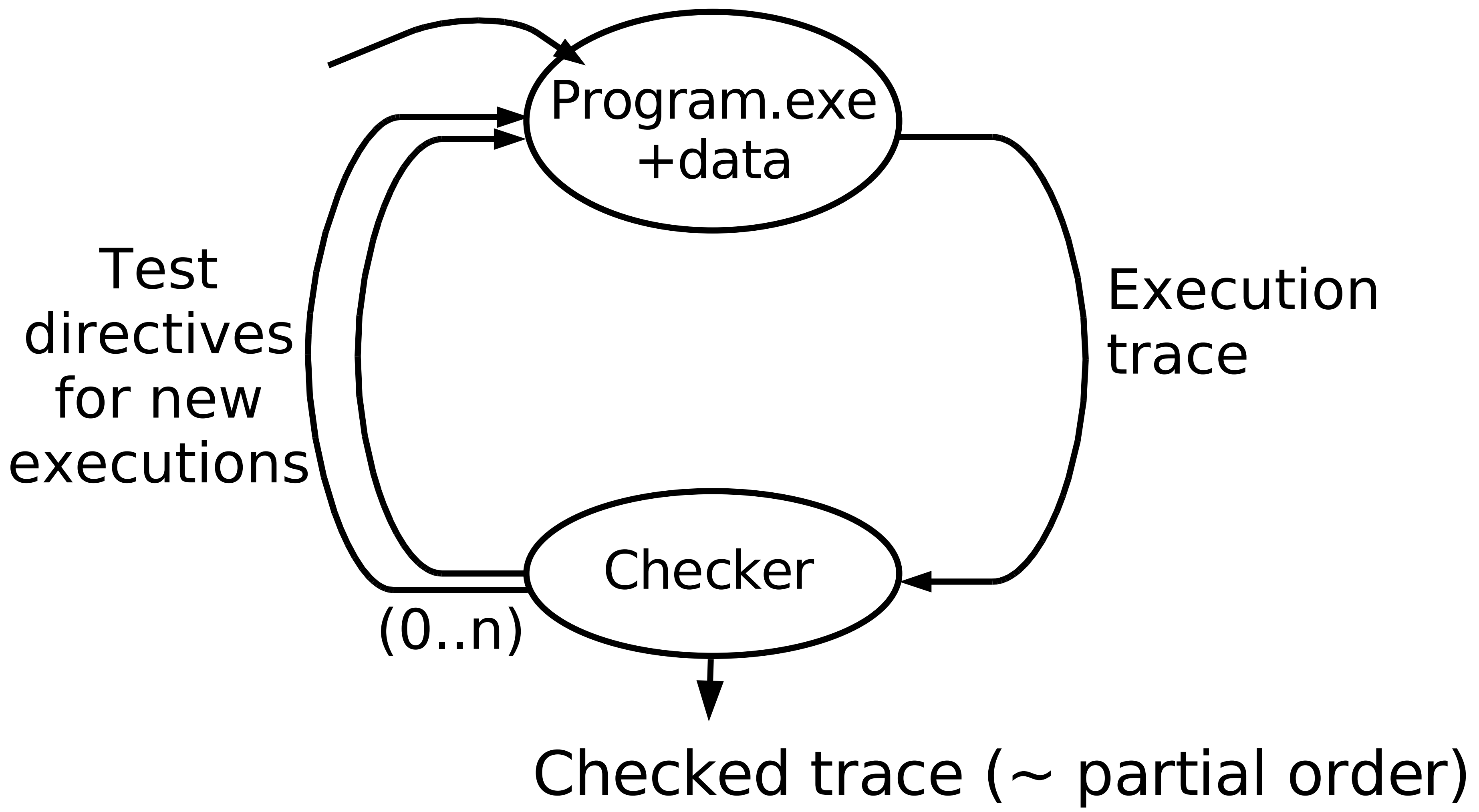
Principle of the Approach

Data is **fixed**; **Delays** are **fixed**;
we generate schedulings

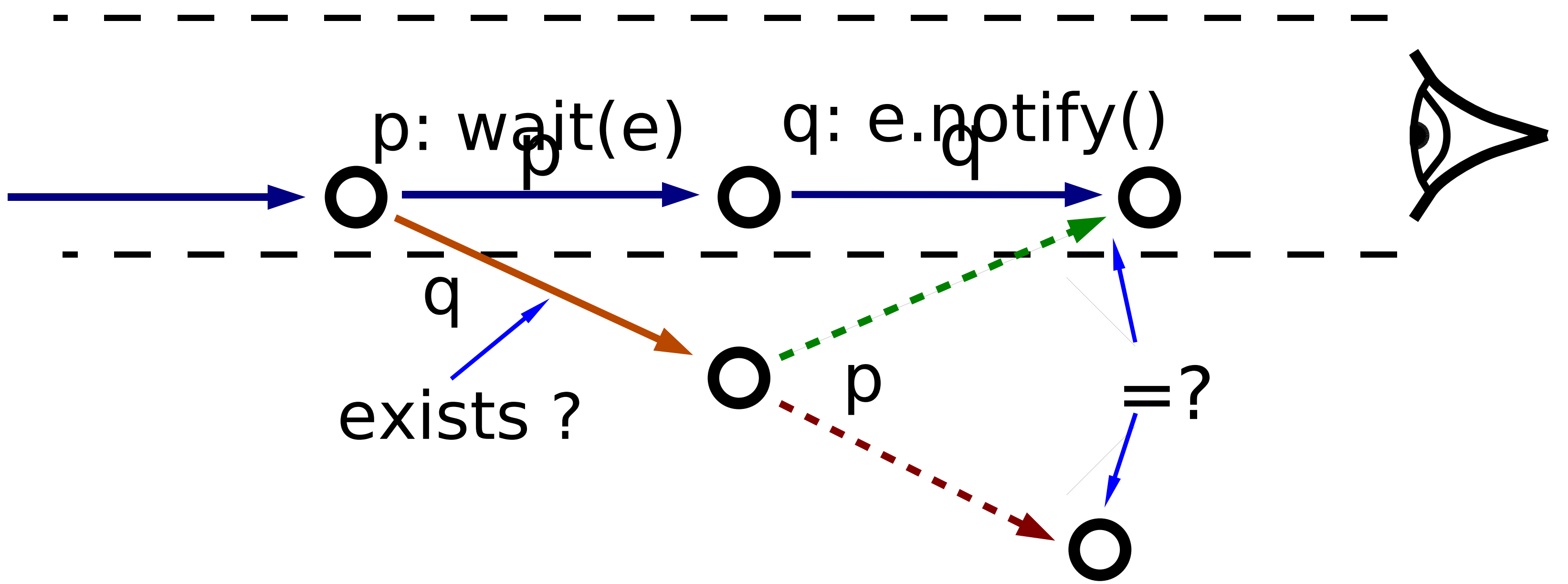


Use of Dynamic Partial Order Reductions
(presented by C.Flanagan, P.Godefroid
at POPL'05)

Cyclic Generation



Checker: Observing Traces



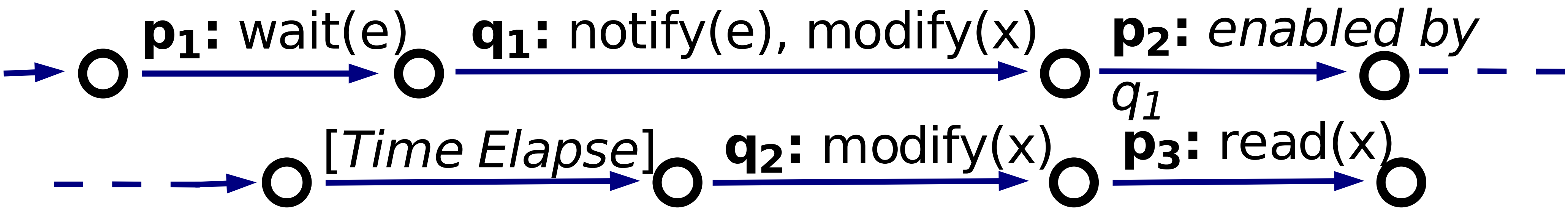
Goal:
Guess if transitions are dependent by observation of their behavior

Checker: Action Dependencies

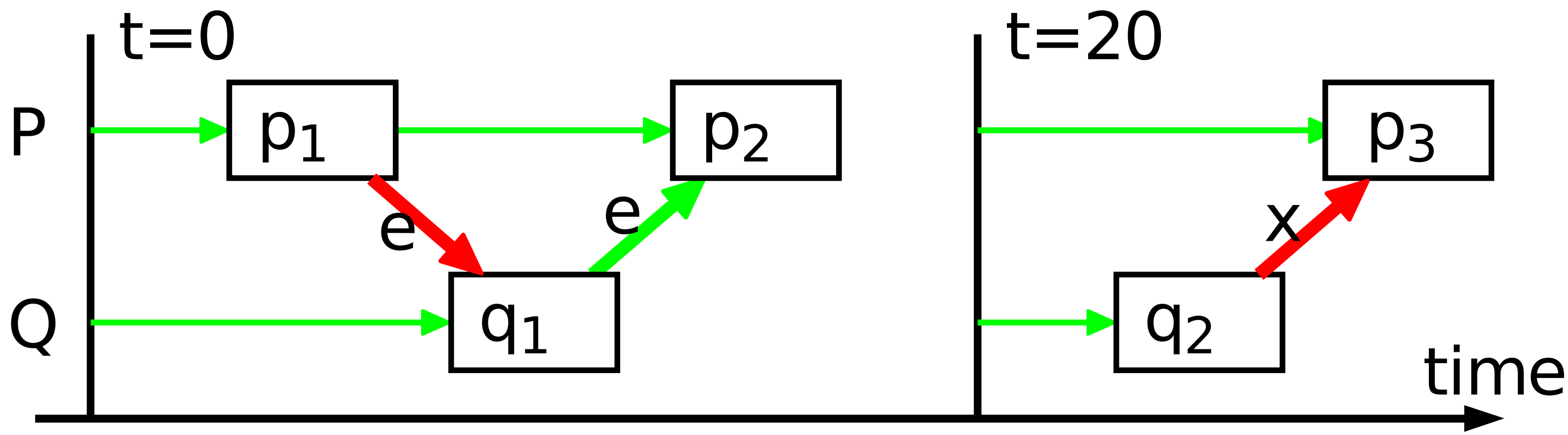
- Independent <=> order is irrelevant
- Dependency cases for SystemC:
  - Variables (or memory locations):
    - Two **`write`** (`T[12]=1` and `T[12]=2`)
    - One **`write`** and one **`read`** (`x=1` and `f(x)`)
  - Events:
    - One **`notify`** and one **`wait`**
    - In some cases: two **`notify`** (consequences on the computed partial order)

Checker: Dynamic Dependency Graph

Execution Trace:



Dynamic Dependency Graph:

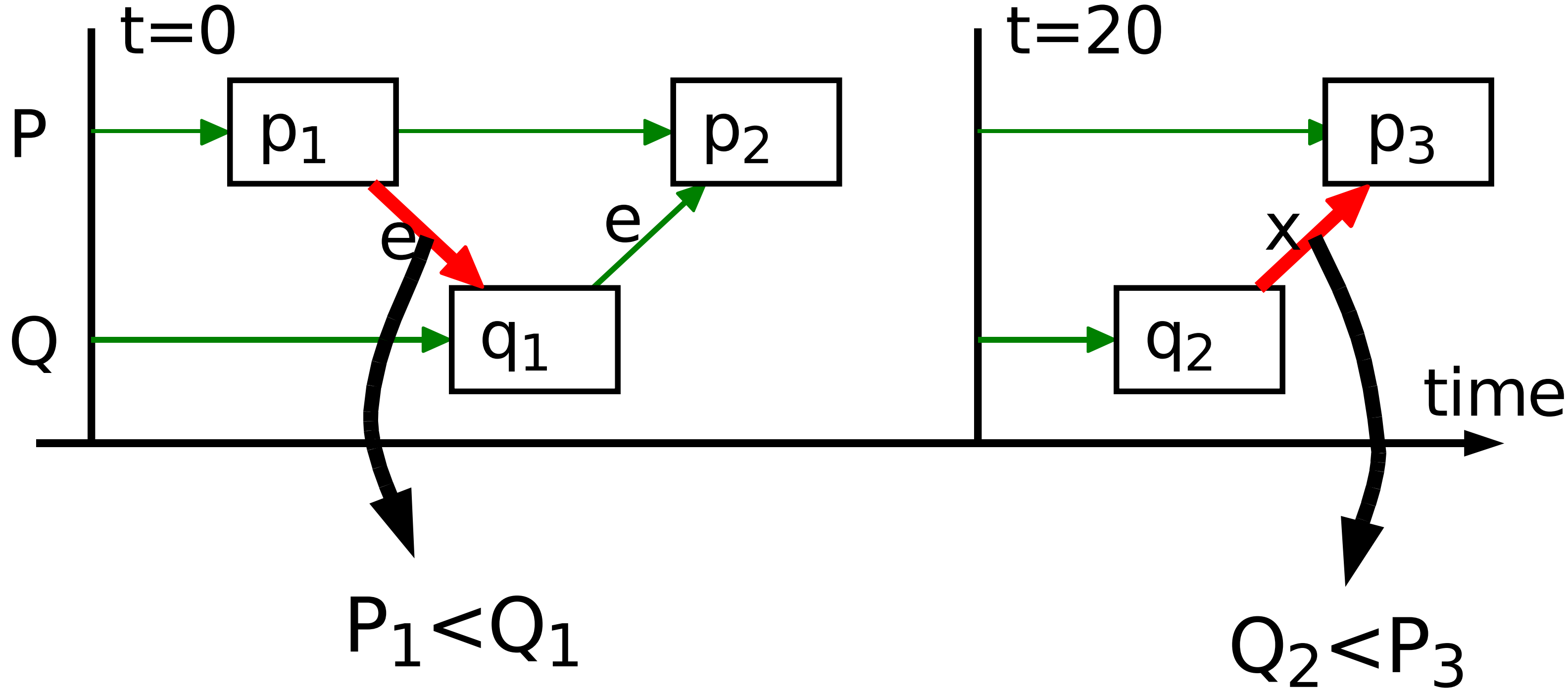


Green arrows: dependent but **not permutable**

Red arrows: dependent and **permutable**

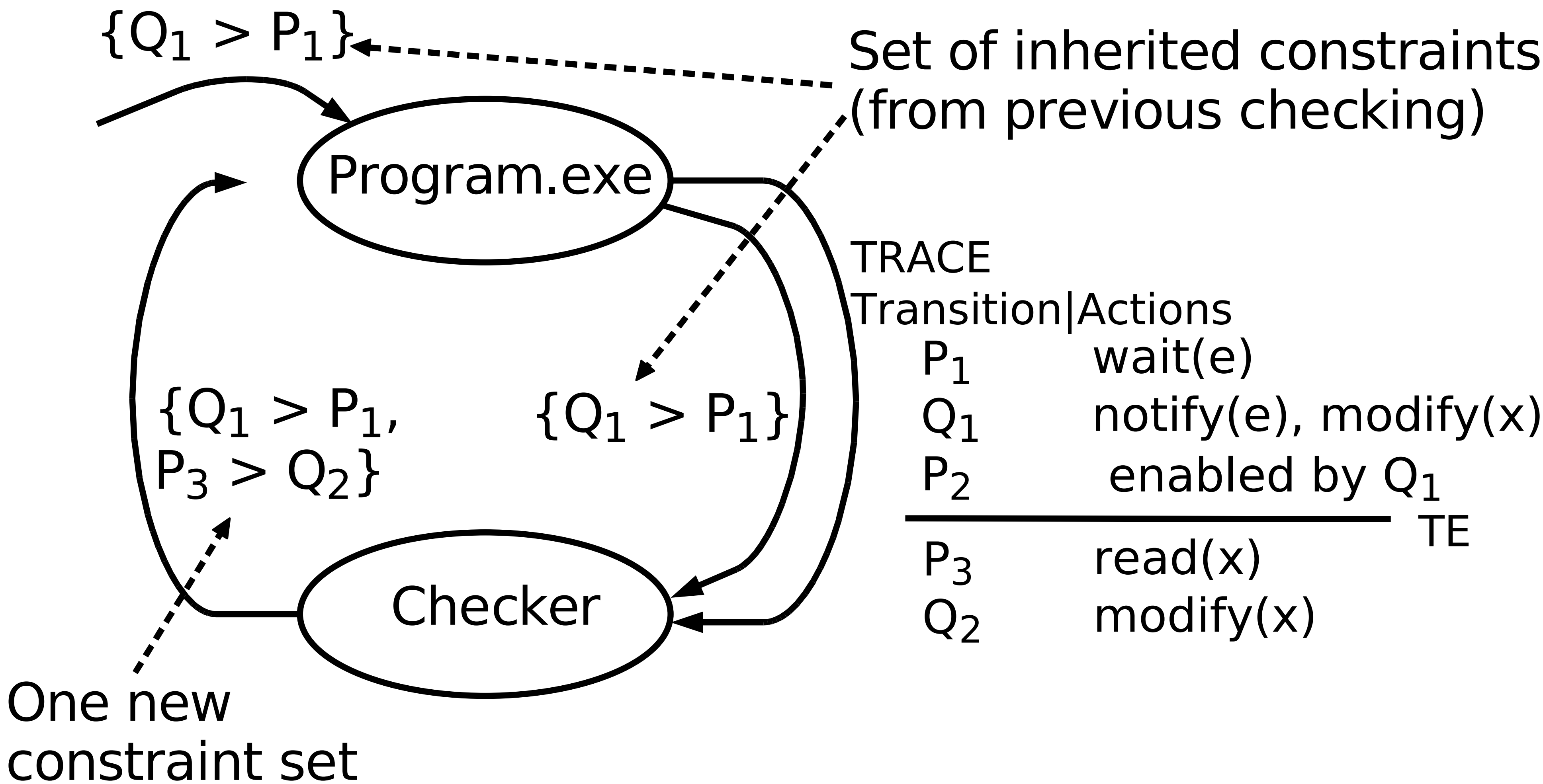
Checker: Scheduling Constraint

Generation of 1 new test directive
for each red arrows



p_i<q_j: **i**-th execution of process **p** before **j**-th execution of process **q**

Cyclic Generation with Scheduling Constraints

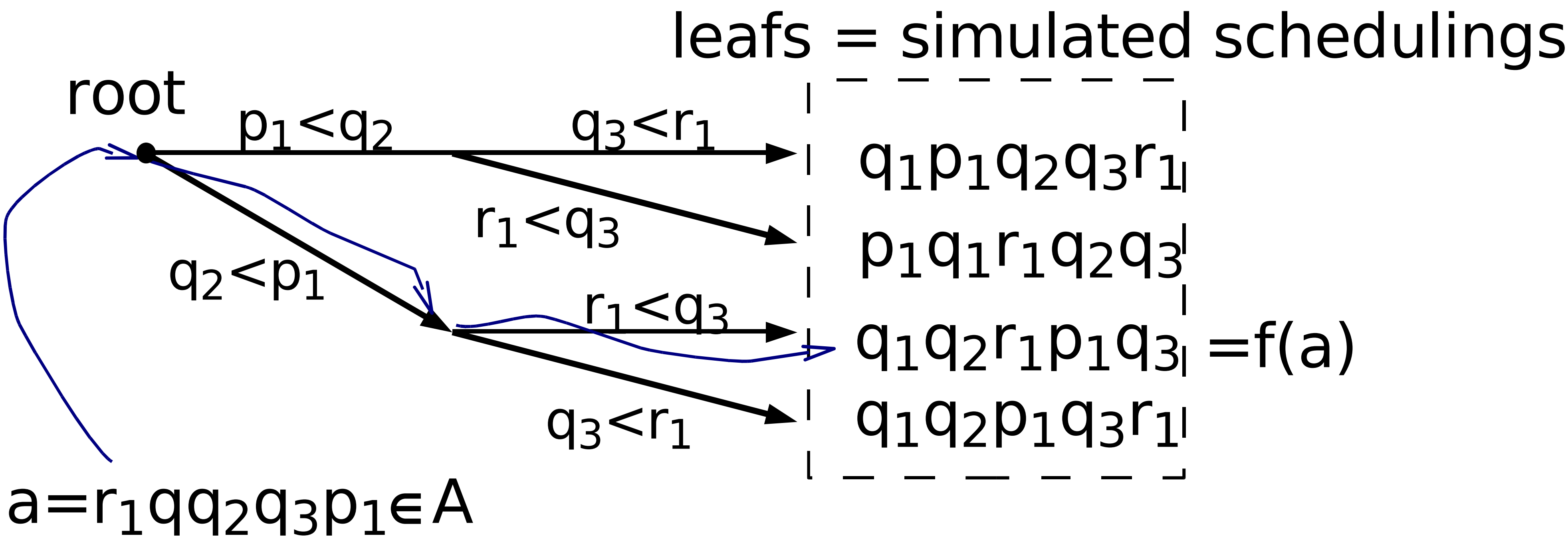


Property Guaranteed by this Method

- **A:** Set of all possible executions (for one data)
- **G:** Set of generated executions (for the same data)

- **Property:** For all ***a*** in **A**, there exists **g** in **G** that differs only by the order of independent transitions.
- **Consequences on coverage:**
  - Full code accessibility for each process
  - All Dead-locks found

Proof Hint: Constraint Trees



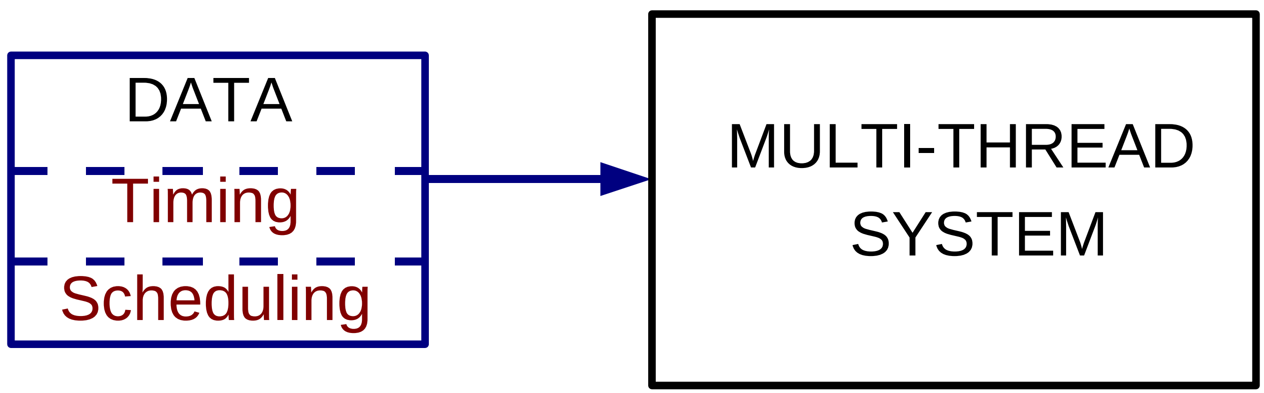
- Define a function f from A to G
- a and f(a) differ only by the order of independent transitions.

Outline

- Context: modeling of SoCs in SystemC-TLM
- Our Problem: managing scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

Principle of the Approach

Data is **fixed**; **Delays** are **bounded**;
we generate schedulings and timings



We deduce linear timing constraints from schedulings constraints, and solve them

What we want to generate

```
void top::P() {
   lwait(3,2); //t1
   wait(e);
   lwait(40,10); //t2
   if (x) cout << "Ok\n";
   else cout << "Ko\n";}
```

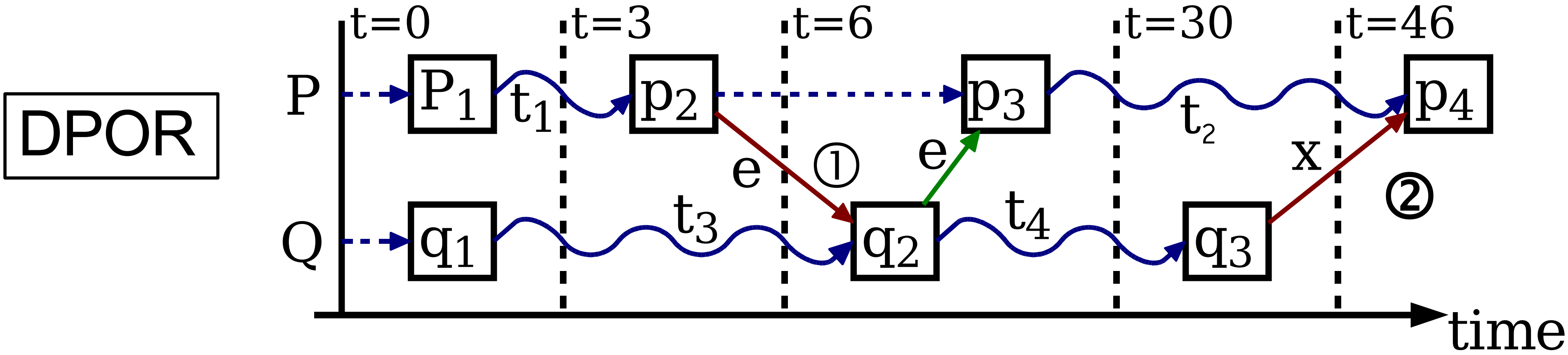
||

```
void top::Q() {
   lwait(6,2); //t3
   e.notify();
   x = 0;
   lwait(24,6); //t4
   x = 1;}
```

- 3 possible executions again:
  - With $t_1 \rightarrow 3$, $t_2 \rightarrow 40$, $t_3 \rightarrow 6$, $t_4 \rightarrow 24$: **Ok**
  - With $t_1 \rightarrow \mathbf{5}$, $t_2 \rightarrow 40$, $t_3 \rightarrow \mathbf{4}$, $t_4 \rightarrow 24$: **dead-lock**
  - With $t_1 \rightarrow 3$, $t_2 \rightarrow \mathbf{30}$, $t_3 \rightarrow 6$, $t_4 \rightarrow \mathbf{30}$: **Ko** possible

Example of Timing Generation

- Dynamic Dependency Graph:



timing: $t_1 \rightarrow 3$, $t_2 \rightarrow 40$, $t_3 \rightarrow 6$, $t_4 \rightarrow 24$

Two Linear Programs to solve:

① q_2 before p_2: $t_3 \leq t_1$, $t_1 \in [1,5]$, $t_3 \in [4,8]$

② p_2 before q_2: $t_3 \geq t_1$, $t_1 \in [1,5]$, $t_3 \in [4,8]$

p_4 before q_3: $t_2 \leq t_4$, $t_2 \in [30,50]$, $t_4 \in [18,30]$

LP

Constraints Generation

- Symbolic date of a transition p_i
  - If enabled by a transition q_j (notification):
    - $sdate(p_i) = sdate(q_j)$
  - If follows a **`lwait(T)`** instruction
    - $sdate(p_i) = sdate(p_{i-1}) + X$
      with X: new variable
- For each scheduling constraint "p_i before q_j":
  - Timing constraint: $sdate(p_i) \leq sdate(q_j)$
- Range of time variables: $T \pm \Delta$

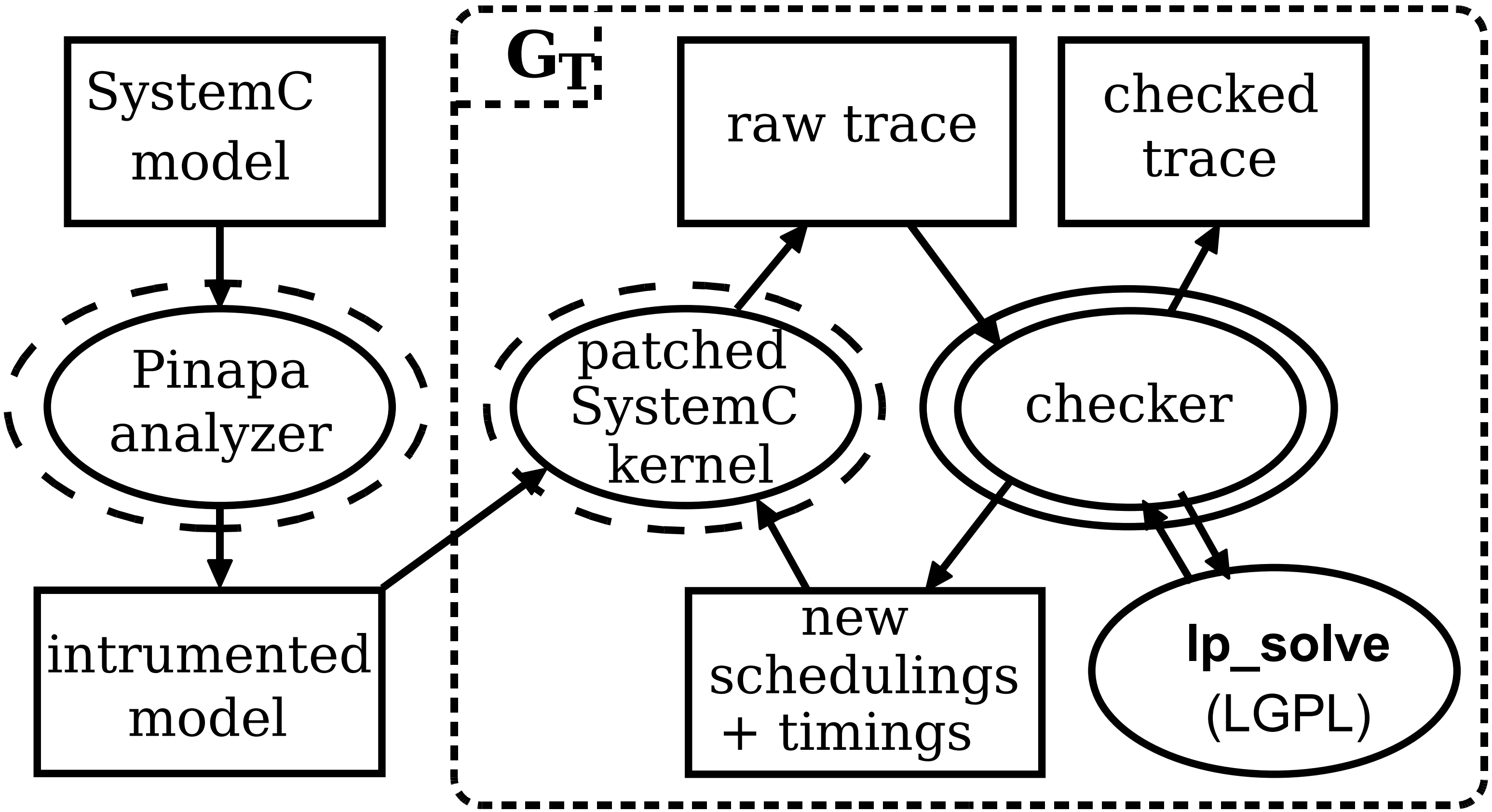
Constraints Solving

- We get a linear program with:
  - 1 variable per `lwait` call
  - 1 constraint per pair of dependent permutable transitions (+ variable ranges)
  - Lots of null coefficients
- We need to exhibit a solution, not only emptyness
- **Solvable without abstraction** using the Simplex Algorithm (first phase only)

Outline

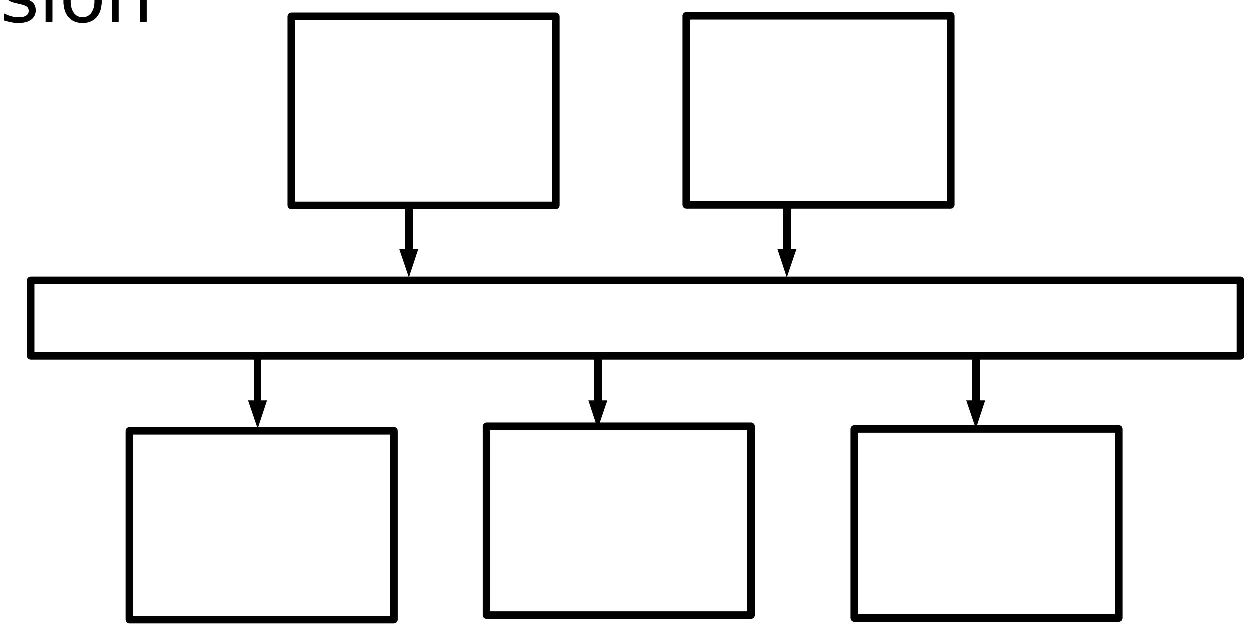
- Context: modeling of SoCs in SystemC-TLM
- Our Problem: managing scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

The Tool Chain



Industrial Case Study: LCMPEG

- Part of a Set-Top Box, from STM
- 5 components, runs of 150 transitions, with long sections of sequential code (~50klines)
- At least 2^40 possible schedulings for the timed version



Case Study: Results

- Fixed Delays:
  - 128 schedulings, 1 min 08 sec
  - overhead: 20% (time spent in checker)
- Loose Delays +/- 20%:
  - 3584 executions, 35 min 11 sec
  - overhead: 33%
- Untimed version:
  - About 2^32 executions needed, failed.

Conclusion of the Case Study

- Works
- Harder for loosely timed TL models because of the complexity of the state space
- Well adapted to abstract TLM models which are asynchronous
- Light tool: no explicit extraction of an abstract formal model, no state comparison, ...

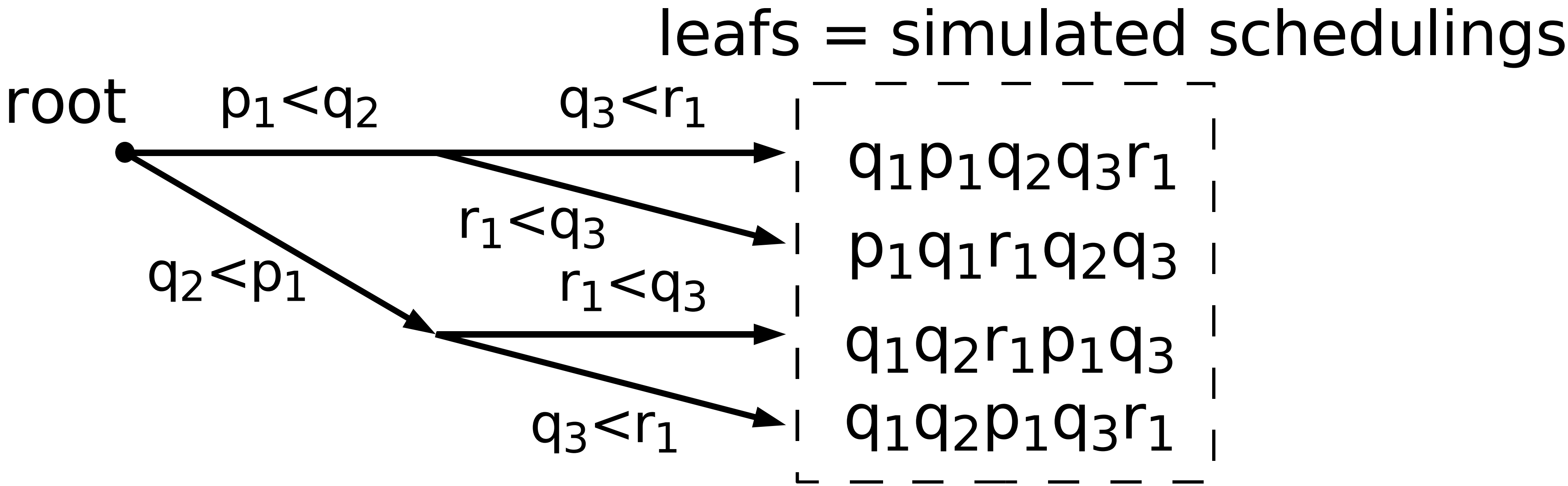
Outline

- Context: modeling of SoCs in SystemC-TLM
- Our Problem: managing scheduling and timing indeterminism
- Covering the valid schedulings
- Covering the valid timings
- Implementation and case study
- Current and further works

Avoid more redundant executions

- Still not perfect: more executions than equivalence classes
  - dead leafs in the constraint tree
  - equivalent leafs in the constraint tree
- Cannot be perfect: counter example exists!
- Can be improved
  - Heuristics in checker and scheduler
  - Detecting dynamically equivalent leafs
- Other solution: try to apply “net unfolding”

Constraint Trees



Better Dependency Analysis: Persistent Events

- Process A: v = 1; e.notify();
- Process B: if (!v) wait(e); v = 0;

- Consequence: useless simulations
- Solution:
  - new class pevent with methods wait, notify and reset
  - extending dependency analysis
- Result: from 128 to 32 generated schedulings for the LCMPEG

Using high level synchronization mechanisms

- Other structures:
  - Variants of persistent events
  - Generic Arbiter
  - Hash table (cf indexer benchmark)
- Should dependency information be included in specifications of components?
- Models can be design in a way such that thay are easier to validate

Thank you for your attention.

Demonstration:
LCMPEG with fixed delays and persistent events

Parallelization of the scheduling & timing generator

- independent subtaskes
- can be run on distant machine