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

Context:

- TLM
- RTL
... 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:

```c
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:

```cpp
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:

```c
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";
}
```

```c
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

```cpp
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;
}
```

- 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

```cpp
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 $t_1 \rightarrow 5$, $t_2 \rightarrow 40$, $t_3 \rightarrow 4$, $t_4 \rightarrow 24$: dead-lock
  - With $t_1 \rightarrow 3$, $t_2 \rightarrow 30$, $t_3 \rightarrow 6$, $t_4 \rightarrow 30$: 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 ⇒ 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

Test directives for new executions

Program.exe +data

Execution trace

(0..n)

Checker

Checked trace (~ partial order)
Checker: Observing Traces

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

Claude Helmstetter, November 2006
Checker: Action Dependencies

- Independent $\iff$ 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:

\( p_1: \text{wait}(e) \quad q_1: \text{notify}(e), \text{modify}(x) \)

\[ \text{Time Elapse} \]

\( p_2: \text{enabled by } q_1 \)

\( p_3: \text{read}(x) \)

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

$\text{P}_1 < \text{Q}_1$

$p_i < q_j$: $i$-th execution of process $p$ before $j$-th execution of process $q$
Cyclic Generation with Scheduling Constraints

Set of inherited constraints (from previous checking)

Program.exe

One new constraint set

Checker

{Q_1 > P_1, P_3 > Q_2}

{Q_1 > P_1}

Checker

{Q_1 > P_1}

TRACER
Transition|Actions
\begin{align*}
P_1 & \quad \text{wait}(e) \\
Q_1 & \quad \text{notify}(e), \text{modify}(x) \\
P_2 & \quad \text{enabled by } Q_1 \\
P_3 & \quad \text{read}(x) \\
Q_2 & \quad \text{modify}(x) \\
\end{align*}
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.

\[ a = r_1 q q_2 q_3 p_1 \in A \]

Leads to simulated schedulings:

\[ q_1 p_1 q_2 q_3 r_1 \]
\[ p_1 q_1 r_1 q_2 q_3 \]
\[ q_1 q_2 r_1 p_1 q_3 \]
\[ q_1 q_2 p_1 q_3 r_1 \]
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

```cpp
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 \( t_1 \rightarrow 5, t_2 \rightarrow 40, t_3 \rightarrow 4, t_4 \rightarrow 24 \): dead-lock
  - With \( t_1 \rightarrow 3, t_2 \rightarrow 30, t_3 \rightarrow 6, t_4 \rightarrow 30 \): Ko possible
Example of Timing Generation

• Dynamic Dependency Graph:

```
dt: t₁ \rightarrow 3, t₂ \rightarrow 40, t₃ \rightarrow 6, t₄ \rightarrow 24
```

Two Linear Programs to solve:

1. \( q₂ \) before \( p₂ \): \( t₃ \leq t₁ \), \( t₁ \in [1,5] \), \( t₃ \in [4,8] \)

2. \( p₂ \) before \( q₂ \): \( t₃ \geq t₁ \), \( t₁ \in [1,5] \), \( t₃ \in [4,8] \)

\( p₄ \) before \( q₃ \): \( t₂ \leq t₄ \), \( t₂ \in [30,50] \), \( t₄ \in [18,30] \)
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 \texttt{lwai}t($T$) 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

SystemC model

Pinapa analyzer

intrumented model

GT

patched SystemC kernel

raw trace

checked trace

new schedulings + timings

lp_solve (LGPL)
Industrial Case Study: LCMPEG

- Part of a Set-Top Box, from STM
- 5 components, runs of 150 transitions, with long sections of sequential code (~50k lines)
- 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

\[ \text{leafs} = \text{simulated schedulings} \]

root \quad p_1 < q_2 \quad q_3 < r_1

\quad q_2 < p_1 \quad r_1 < q_3

\quad r_1 < q_3 \quad r_1 < q_3

\quad q_3 < r_1

\quad q_1 p_1 q_2 q_3 r_1

\quad p_1 q_1 r_1 q_2 q_3

\quad q_1 q_2 r_1 p_1 q_3

\quad q_1 q_2 p_1 q_3 r_1

Claude Helmstetter, November 2006
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 they are easier to validate
Thank you for your attention.
Demonstration: LCMPEG with fixed delays and persistent events
Parallelization of the scheduling & timing generator

- independent subtasks
- can be run on distant machine