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, state, media)
    - 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

<table>
<thead>
<tr>
<th>In MMM</th>
<th>In SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td>process</td>
<td>sc_module</td>
</tr>
<tr>
<td>medium</td>
<td>sc_channel</td>
</tr>
<tr>
<td>state medium</td>
<td>sc_channel</td>
</tr>
<tr>
<td>quantity</td>
<td>sc_channel</td>
</tr>
<tr>
<td>netlist</td>
<td>sc_channel, sc_main</td>
</tr>
<tr>
<td>port</td>
<td>sc_port</td>
</tr>
<tr>
<td>interface</td>
<td>sc_interface</td>
</tr>
</tbody>
</table>
Overall Framework

Scoreboard

Process Info & Functions

Manager

P0

P1

M

C0
Simulation Algorithm

- Alternating running phases
  - Process phase
  - Manager phase

```
Manager

Scoreboard

P0

P1

C0

…

i1

i1+1

i2

i2+1

i3

DeltaCycle
```
Simulation Algorithm (2)

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

```c
process P {
    ...
    void thread()
    {
        ...
        await(C;T;S);...
        ...
        portA.IFC1();
        ...
        Label1: x = 5;
        ...
    }
}
```
interface reader extends Port{
  update int read();
  eval int n();
}

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

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();
        }
      }
    }
  }
}

P0, P1, C0 prototype

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();

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

- **System using GlobalTime quantity**

  ![Diagram](image)

- **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

wait for Manager phase

do await-intfc scheduling

call quantity resolution functions

go back to Process phase

cleanup by Q::postcond()

Q1::resolve()

Q2::resolve()

Q3::resolve()

Fix-point computation
Simulation Result

A sample of simulation result

Read

Write

Time
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

\[ \text{constraint}\{ \]

// mutual exclusion between P0 and P1

\[ \text{ltl}( G( \text{beg}(P0, \text{M.write}) \rightarrow ( (\! \text{beg}(P1, \text{M.write})) \text{ U end}(P0, \text{M.write}) ) ) ) ); \]

\[ \text{ltl}( G( \text{beg}(P1, \text{M.write}) \rightarrow ( (\! \text{beg}(P0, \text{M.write})) \text{ U end}(P1, \text{M.write}) ) ) ) ); \]

\} \} \]

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

<table>
<thead>
<tr>
<th>Without LTL Enforcement</th>
<th>With LTL Enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor&gt; c read BEGIN</td>
<td>monitor&gt; c read BEGIN</td>
</tr>
<tr>
<td>monitor&gt; P0 write BEGIN</td>
<td>monitor&gt; P0 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; P1 write BEGIN</td>
<td>monitor&gt; P0 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; c read END</td>
<td>monitor&gt; c read END</td>
</tr>
<tr>
<td>monitor&gt; P0 write END</td>
<td>monitor&gt; c read END</td>
</tr>
<tr>
<td>monitor&gt; P1 write END</td>
<td>monitor&gt; P0 write END</td>
</tr>
<tr>
<td>monitor&gt; c read END</td>
<td>monitor&gt; P0 write END</td>
</tr>
<tr>
<td>monitor&gt; P0 write BEGIN</td>
<td>monitor&gt; P0 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; P1 write BEGIN</td>
<td>monitor&gt; P0 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; c read BEGIN</td>
<td>monitor&gt; P1 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; P0 write BEGIN</td>
<td>monitor&gt; c read BEGIN</td>
</tr>
<tr>
<td>monitor&gt; P1 write BEGIN</td>
<td>monitor&gt; P0 write BEGIN</td>
</tr>
<tr>
<td>monitor&gt; c read END</td>
<td>monitor&gt; P1 write END</td>
</tr>
<tr>
<td>monitor&gt; P0 write END</td>
<td>monitor&gt; c read END</td>
</tr>
<tr>
<td>monitor&gt; P1 write END</td>
<td>monitor&gt; P0 write END</td>
</tr>
</tbody>
</table>

---

violation

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)

<table>
<thead>
<tr>
<th>Opt Tech</th>
<th>Sim. Time(s)</th>
<th>Cycle/Second*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>7276</td>
<td>9.16K</td>
</tr>
<tr>
<td>Native SystemC</td>
<td>22.7</td>
<td>2.94M</td>
</tr>
</tbody>
</table>

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.

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

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

\[
\text{await}(guard; ; ) \ \{\text{critical section}\} \text{ or if } (guard) \ \{\text{critical section}\}
\]
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

P1 → M ← P2

Events:
- e0,
- e1,
- ...
- read()
- e2,
- e3,
- ...

Task → CPU → MEM

Events:
- e0',
- e1',
- ...
- readMEM()
- e2',
- e3',
- ...

synch
Simultaneity constraints (2)

Elaborate constraints

Construct synch events equivalent classes:

- synch(e₁, e₂);
- synch(e₁, e₃);
- synch(e₄, e₅);

Annotate equivalent classes info:

- e₁ ~ group 0
- e₄ ~ group 1
- ...

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

Generate SystemC code

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

Case Study (2)

Picture-in-Picture behavior simulation result

<table>
<thead>
<tr>
<th>Option</th>
<th>Simulation Time (s)</th>
<th>Cycle/Second*</th>
<th>Overall Speedup</th>
<th>Speedup by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>7276</td>
<td>9.16K</td>
<td>1</td>
<td>---</td>
</tr>
<tr>
<td>MC</td>
<td>1797</td>
<td>37.1K</td>
<td>4</td>
<td>MC: 4</td>
</tr>
<tr>
<td>MC/NER</td>
<td>89.26</td>
<td>747K</td>
<td>80</td>
<td>NER: 20</td>
</tr>
<tr>
<td>MC/NER/IC</td>
<td>20.29</td>
<td>3.29M</td>
<td>359</td>
<td>IC: 4.5</td>
</tr>
<tr>
<td>Native SystemC</td>
<td>22.7</td>
<td>2.94M</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

- 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

<table>
<thead>
<tr>
<th># of Simultaneity Constraints</th>
<th>Handling Overhead *</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2.9%</td>
</tr>
<tr>
<td>16</td>
<td>2.9%</td>
</tr>
<tr>
<td>32</td>
<td>3.4%</td>
</tr>
<tr>
<td>64</td>
<td>4.0%</td>
</tr>
</tbody>
</table>

*: 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 tools. In general it could be huge.

- We applied a few techniques to minimize the overhead. From the simulation result, it shows 4X to 20X speedup of individual techniques. Combine the techniques together, they eliminate all overhead.

Efficient SystemC-based Metropolis Simulator!
Questions?