Transaction-Level Models of Systems-on-a-Chip
Can they be Fast, Correct and Faithful?

Matthieu Moy

Laboratoire d’Informatique du Parallelisme
Lyon, France

February 2018
About me

2005  Ph.D: formal verification of SoC models (ST/Verimag)
2006  Post-doc: security of storage (Bangalore, Inde)
2006  Assistant professor, Verimag / Ensimag
       Work on SoC models & abstract interpretation
2014  HDR: High-Level models for Embedded Systems
2017  New CASH team leader, LIP / UCBL
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion
Modern Systems-on-a-Chip
Modern Systems-on-a-Chip

Software

Hardware

Matthieu Moy (LIP)  Transaction-Level Models of SoCs  February 2018
Hardware/Software Design Flow

Traditional Design-Flow

- Specification, Algorithm
- RTL Design
- Synthesis
- Factory
- Software Development
- Integration
- Validation
Hardware/Software Design Flow

Traditional Design-Flow

- Specification, Algorithm
- RTL Design
- Synthesis
- Factory
- Software Development
- Integration
- Validation

Time

Cost > 1,000,000 $!
Hardware/Software Design Flow

Traditional Design-Flow

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation

Transaction-Level Model based

1. Specification, Algorithm
2. RTL Design
3. Synthesis

Matthieu Moy (LIP)
Hardware/Software Design Flow

Traditional Design-Flow

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation

Transaction-Level Model based

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Software Development

Matthieu Moy (LIP)
Hardware/Software Design Flow

Traditional Design-Flow

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation

Transaction-Level Model based

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. TLM Model
5. Software Development
Hardware/Software Design Flow

Traditional Design-Flow

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation

Transaction-Level Model based

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation
Hardware/Software Design Flow

Traditional Design-Flow

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation

Transaction-Level Model based

1. Specification, Algorithm
2. RTL Design
3. Synthesis
4. Factory
5. Software Development
6. Integration
7. Validation
8. TLM Model

Time

Gain
The Transaction Level Model: Principles and Objectives

A high level of abstraction, that appears early in the design-flow
The Transaction Level Model: Principles and Objectives

A high level of abstraction, that appears early in the design-flow

- A **virtual prototype** of the system, to enable
  - Early software development
  - Integration of components
  - Architecture exploration
  - Reference model for validation

- **Abstract** implementation details from RTL
  - Fast simulation ($\simeq 1000x$ faster than RTL)
  - Lightweight modeling effort ($\simeq 10x$ less than RTL)
Content of a TLM Model

A first definition

- Model what is **needed for Software Execution**:
  - Processors
  - Address-map
  - Concurrency

- ... and **only that**.
  - No micro-architecture
  - No bus protocol
  - No pipeline
  - No physical clock
  - ...
An example TLM Model

- CPU
- ITC
- VGA
- Timer
- Data RAM
- Instruction RAM
- GPIO

process = C++ code
Performance of TLM

- Pure RTL: 1 hour
- RTL + cosimulation: 3 minutes
- TLM: 3 seconds
- HW emulation: 1 second

Simulation time (second) logarithmic scale
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development

Unmodified Software

SoCs and TLM

Compilation
Verification
Extra-functional
Conclusion
Uses of Functional Models

Reference for Hardware Validation

Unmodified Software

Virtual Prototype for Software Development
Content of a TLM Model

A richer definition

- **Timing information**
  - May be needed for Software Execution
  - Useful for Profiling Software

- **Power and Temperature**
  - Validate design choices
  - Validate power-management policy
Use of Extra-Functional Models
Timing, Power consumption, Temperature Estimation
Use of Extra-Functional Models
Timing, Power consumption, Temperature Estimation
Use of Extra-Functional Models
Timing, Power consumption, Temperature Estimation

Unmodified Power/Temperature-Aware Software

Estimated ≈ Actual
Summary: Expected Properties of TLM Programs

SystemC/TLM Programs should

- Simulate **fast**,  
- Satisfy **correctness** criterions,  
- Reflect **faithfully** functional and extra-functional properties of the actual system.
<table>
<thead>
<tr>
<th>SoC and TLM</th>
<th>Compilation</th>
<th>Verification</th>
<th>Extra-functional</th>
<th>Conclusion</th>
</tr>
</thead>
</table>

**Outline**

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling

2. Compilation of SystemC/TLM

3. Verification of SystemC/TLM

4. Extra-Functional Properties in TLM

5. Conclusion
SystemC: Simple Example

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;

    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};

int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    // Instantiate modules ...
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // ... and bind them together
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS);
    return 0;
}
```
Compiling SystemC

$ g++ example.cpp -lsystemc
$ ./a.out

... end of section?
Compiling SystemC

$ g++ example.cpp -lsystemc
$ ./a.out

But ...

- C++ compilers cannot do SystemC-aware optimizations
- C++ analyzers do not know SystemC semantics
Compilation of SystemC/TLM
- Front-end
- Optimization and Fast Simulation
In this talk: **Front-end** = “Compiler front-end” (AKA “Parser”)

**SystemC Front-End**

- **SystemC** → **Front end** → **Intermediate Representation** → **Back end**

**Intermediate Representation** = Architecture + Behavior
SystemC Front-Ends

- When you *don’t* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...
SystemC Front-Ends

**When you *don’t* need a front-end:**
- Main application of SystemC: Simulation
- Testing, run-time verification, monitoring...

⇒ No reference front-end available on
http://www.accellera.org/
SystemC Front-Ends

- When you *don’t* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...
  ⇒ No reference front-end available on http://www.accellera.org/

- When you *do* need a front-end:
  - Symbolic formal verification, High-level synthesis
  - Visualization
  - Introspection
  - SystemC-specific Compiler Optimizations
  - Advanced debugging features
Challenges and Solutions with SystemC Front-Ends

1. **C++ is complex (e.g. *clang* $\approx 200,000$ LOC)**

2. **Architecture built at runtime, with C++ code**

```c++
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};

int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);
    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```
Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. clang $\approx 200,000$ LOC)
   $\Rightarrow$ Write a C++ front-end or reuse one (g++, clang, EDG, ...)

2. Architecture built at runtime, with C++ code
   $\Rightarrow$ Analyze elaboration phase or execute it

```c
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};

int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);
    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Matthieu Moy (LIP)
Challenges and Solutions with SystemC Front-Ends

1. **C++ is complex (e.g. clang ≈ 200,000 LOC)**
   \[\Rightarrow\] **Write** a C++ front-end or **re**use one (g++, clang, EDG, ...)

2. **Architecture built at runtime, with C++ code**
   \[\Rightarrow\] **Analyze** elaboration phase or **execute** it

```c++
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```c++
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);
    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

**Static Approaches**

**Dynamic Approaches**
Dealing with the architecture

When it becomes tricky...

```c
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j] = new Node(...);
            ...
        }
    }
    sc_start(100, SC_NS);
    return 0;
}
```
Dealing with the architecture
When it becomes tricky...

- **Static** approach: cannot deal with such code
- **Dynamic** approach: can extract the architecture for individual instances of the system

```c
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j] = new Node(...);
        }
    }
    sc_start(100, SC_NS);
    return 0;
}
```
Dealing with the architecture

When it becomes very tricky…

```c
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```
Dealing with the architecture

When it becomes very tricky...

- One can unroll the loop to let $i$ become constant,
- Undecidable in the general case.

```c
void compute(void) {
    for (int i = 0; i < n; i++) {
        ports[i].write(true);
    }
    ...
}
```
**The beginning: Pinapa**

AKA “my Ph.D’s front-end”

- **Pinapa’s principle:**
  - Use GCC’s C++ front-end
  - Compile, dynamically load and execute the elaboration (\texttt{sc\_main})

- **Pinapa’s drawbacks:**
  - Uses GCC’s internals (hard to port to newer versions)
  - Hard to install and use, no separate compilation
  - Ad-hoc match of SystemC constructs in AST
  - AST Vs SSA form in modern compilers
LLVM: Low Level Virtual Machine

- **C**
- **C++**
- **...**

Clean API
- Clean SSA intermediate representation
- Many tools available
LLVM: Low Level Virtual Machine

- Clean API
- Clean SSA intermediate representation
- Many tools available

Matthieu Moy (LIP)
PinaVM: Enriching the bitcode

SystemC

Compilation
(llvm-g++, llvm-link)

LLVM bitcode

Execute elaboration

Identify SC constructs

Architecture

bitcode++

Intermediate Representation
PinaVM: Enriching the bitcode

SystemC

Compilation
(llvm-g++, llvm-link)

SystemC construct is still a normal function

%this is fixed

LLVM bitcode

Execute elaboration

Identify SC constructs

%this not known
Cannot compute %port

%port = expr1(%this)
%data = expr2
call write %port, %data

Intermediate Representation

Execute dependencies

Architecture

executeelaboration

...%port = expr1(%this)%data = expr2
SCWrite
- data = ??
- port = ??

...%port = expr1(%this)%data = expr2
SCWrite
- data = \[
\begin{array}{c}
\text{Process } 0 \rightarrow \text{data } d_0 \\
\text{Process } 1 \rightarrow \text{data } d_1
\end{array}
\]
- port = \[
\begin{array}{c}
\text{Process } 0 \rightarrow \text{port } p_0 \\
\text{Process } 1 \rightarrow \text{port } p_1
\end{array}
\]
Summary

- PinaVM relies on **executability** (JIT Compiler) for execution of:
  - elaboration phase (≈ like Pinapa)
  - sliced pieces of code


- Still a prototype, but very few fundamental limitations

- ≈ 3000 lines of C++ code on top of LLVM

- Experimental back-ends for
  - Execution (Tweto)
  - Model-checking (using SPIN)
This section

Compilation of SystemC/TLM
- Front-end
- Optimization and Fast Simulation
Typical Transaction Journey

CPU

Bus

T1

T2

RAM

Call virtual method on socket
Forward method call to target socket
Address decoding
Another virtual method call Forwarded to target socket
Ends-up calling target module's method
Typical Transaction Journey

```
... port.write(addr, data);
...
```

```
status write(addr, data) {
    mem[addr] = data;
}
```
Typical Transaction Journey

```
port.write(addr, data);
```

```
status write(addr, data) {
    mem[addr] = data;
}
```
Typical Transaction Journey

```
.. port.write(addr, data);
...
```

```
status write(addr, data) {
    mem[addr] = data;
}
```

Call virtual method on socket

Forward method call to target socket

Forwarded to target socket

Ends-up calling target module’s method
Many costly operations for a simple functionality
Work-around: backdoor access (DMI = Direct Memory Interface)
  ▶ CPU get a pointer to RAM’s internal data
  ▶ Manual, dangerous optimization
Many costly operations for a simple functionality
- Work-around: backdoor access (DMI = Direct Memory Interface)
  - CPU get a pointer to RAM’s internal data
  - Manual, dangerous optimization

Can a compiler be as good as DMI, automatically and safely?
Basic Ideas

- Do **statically** what can be done **statically** ...
- ... considering “**statically**” = “after elaboration”
- Examples:
  - Virtual function resolution
  - Inlining through SystemC ports
  - Static address resolution
Dealing with addresses *Statically*

```java
status write(addr, data) {
    mem[addr] = data;
}
```
Dealing with addresses *Statically*

```java
port.write(0x5500, data);

status write(addr, data) {
    mem[addr] = data;
}
```
Dealing with addresses **Statically**

```java
... port.write(0x5500, data);
...
```

Get actual port addr from PinaVM

Follow path to bus

CPU

0x0000 0x1000 0x2000 0x3000 0x5000 0x6000

RAM

T1

T2

Get actual port addr from PinaVM

Follow path to bus

Matthieu Moy (LIP)
Dealing with addresses **Statically**

```cpp
port.write(0x5500, data);
```

**Get actual port addr from PinaVM**

**Follow path to bus**

**Address Decoding**

```cpp
status write(addr, data) {
    mem[addr] = data;
}
```
Dealing with addresses *Statically*

```
port.write(0x5500, data);
```

```
status write(addr, data) {
  mem[addr] = data;
}
```
Dealing with addresses **Statically**

```
port.write(0x5500, data);
```

```
status write(addr, data) {
    mem[addr] = data;
}
```

- Get actual port addr from PinaVM
- Follow path to bus
- Address Decoding
- Find target socket at this address
- Find function in target module
- CPU

```
... port.write(0x5500, data);
... 
```

```
0x0000
0x1000
0x2000
0x3000
0x5000
0x6000
```

-T1

- RAM

-Follow path to bus

-Address Decoding

-Find target socket at this address

-Find function in target module

-Matthieu Moy (LIP)
Dealing with addresses *Statically*

Possible optimizations:
- Replace call to `port.write()` with `RAM.write()`
- Possibly inline it
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion
Encoding Approaches

- SystemC
  - Encoding
    - Formal language
      - Existing verifier
        - Yes/No/Maybe
Encoding Approaches

- $T_1 \times T_2 \times T_3 \times \text{Sch}$
  - Synchronous automata + scheduler
- $T_1 \times T_2 \times T_3$
  - Asynchronous product shared variable
- $T_1 \otimes T_2 \otimes T_3$
  - Asynchronous automata

SystemC
Concurrent program

- Dedicated product
- Asynchronous automata
Encoding Approaches

\[
\begin{align*}
T_1 \times T_2 \times T_3 \times \text{Sch} & \quad \text{Synchronous automata + scheduler} \\
T_1 \times T_2 \times T_3 & \quad \text{Asynchronous product shared variable} \\
T_1 \bigotimes T_2 \bigotimes T_3 & \quad \text{Asynchronous automata} \\
T_1 \times T_2 \times T_3 \times \text{Sch} & \quad \text{Asynchronous automata} \\
\end{align*}
\]
Translating a SystemC Program

- **Translation** = Parse the source code, generate an automaton
- **Direct semantics** = Read the specification, instantiate an automaton
Translating a SystemC Program

- **Translation** = Parse the source code, generate an automaton
- **Direct semantics** = Read the specification, instantiate an automaton
Translating a SystemC Program

- **Translation** = Parse the source code, generate an automaton
- **Direct semantics** = Read the specification, instantiate an automaton

User code:
Automatic translation

Scheduler

Transaction-Level Models of SoCs
February 2018
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton
Translating a SystemC Program

- Translation = Parse the source code, generate an automaton
- Direct semantics = Read the specification, instantiate an automaton
The SystemC scheduler

- **Non-preemptive** scheduler
- **Non-deterministic** processes election

Select process

Run \[\rightarrow\] Init \[\rightarrow\] Update \[\rightarrow\] Time elapse

(+ 1 automaton per process to reflect its state)
Encoding Approaches

- Synchronous automata + scheduler: $T_1 \times T_2 \times T_3 \times \text{Sch}$
- Asynchronous product shared variable: $T_1 \times T_2 \times T_3$
- Asynchronous automata: $T_1 \times T_2 \times T_3 \times \text{Sch}$
- Dedicated product: $T_1 \times T_2 \times T_3$
Encoding Approaches

- Synchronous automata + scheduler: $T_1 \times T_2 \times T_3 \times \text{Sch}$
- Asynchronous automata: $T_1 \times T_2 \times T_3$
- Asynchronous product shared variable: $T_1 \times T_2 \times T_3$
- Dedicated product: $T_1 \times T_2 \times T_3 \times \text{Sch}$

SystemC Concurrent program
Encoding Approaches

- **Synchronous automata**: $T_1 \times T_2 \times T_3 \times \text{Sch}
- **Asynchronous automata**: $T_1 \otimes T_2 \otimes T_3$
- **SystemC Concurrent program**: $T_1 \times T_2 \times T_3$
- **Asynchronous product shared variable**: $T_1 \times T_2 \times T_3$
- **Dedicated product**

Matthieu Moy (LIP)
Encoding Approaches

\[ T_1 \times T_2 \times T_3 \times \text{Sch} \]

Synchronous automata + scheduler

\[ T_1 \otimes T_2 \otimes T_3 \]

Asynchronous automata

Dedicated product

\[ T_1 \times T_2 \times T_3 \]

Asynchronous product

shared variable

SystemC

Concurrent program

\[ T_1 \times T_2 \times T_3 \times \text{Sch} \]

Asynchronous automata
SystemC to Spin: encoding events

1. notify/wait for event $E^k$:

\begin{align*}
& p::\text{wait}(E^k): \\
& \quad W_p := k \\
& \quad \text{blocked}(W_p == 0)
\end{align*}

\begin{align*}
& p::\text{notify}(E^k): \\
& \quad \forall i \in P \mid W_i == K \\
& \quad W_i := 0
\end{align*}


\[ W_p = k \iff \text{“process } p \text{ is waiting for event } E^k”. \]
SystemC to Spin: encoding time and events

- discrete time
- a deadline variable $T_p$ is attached to each process $p$
  $T_p =$ next execution time for process $p$

```
p::wait(d):
  $T_p := T_p + d$
  blocked($T_p == \min_{i \in P} (T_i)$)
```

“Set my next execution time to now + $d$ and wait until the current execution time reaches it”
SystemC to Spin: encoding time and events

- discrete time
- a deadline variable $T_p$ is attached to each process $p$
  $T_p = \text{next execution time for process } p$

\begin{align*}
\text{\texttt{p::wait(d):}} \quad & \quad T_p := T_p + d \\
& \quad \text{blocked}(T_p == \min_{i \in P} (T_i)) \\
& \quad \text{W}_i == 0
\end{align*}

\text{“Set my next execution time to now + d and wait until the current execution time reaches it”}

\begin{align*}
\text{\texttt{p::wait(E^k):}} \quad & \quad W_p := K \\
& \quad \text{blocked}(W_p == 0)
\end{align*}

\begin{align*}
\text{\texttt{p::notify(E^k):}} \quad & \quad \forall i \in P \mid W_i == K \\
& \quad W_i := 0 \\
& \quad T_i := T_p
\end{align*}
SystemC to Spin: results

![Graph showing results comparison between PinaVM and SPIN 07 for various components and states. The x-axis represents the number of components, and the y-axis represents the number of states. The graph shows a significant increase in states with increasing components for both models, with PinaVM having a smoother curve compared to SPIN 07.]
Encoding Approaches

- **Synchronous automata** + scheduler: $T_1 \times T_2 \times T_3 \times \text{Sch}$
- **Asynchronous automata**
  - Dedicated product: $T_1 \otimes T_2 \otimes T_3$
  - Asynchronous product
    - Shared variable: $T_1 \times T_2 \times T_3$
    - Asynchronous automata: $T_1 \times T_2 \times T_3 \times \text{Sch}$
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion
This section

Extra-Functional Properties in TLM

- Time and Parallelism
- Power and Temperature Estimation
Parallelization of Simulations
Parallelization of Simulations

System-level Simulation Vs HPC
Problems and solutions for parallel execution of SystemC/TLM

1. Execution order imposed by SystemC semantics
2. Concurrent access to shared resources (e.g., $x++$ on a global variable)
Problems and solutions for parallel execution of SystemC/TLM

(1) Execution order imposed by SystemC semantics
(2) Concurrent access to shared resources
   (e.g., \( x++ \) on a global variable)

\( \leadsto \) No 100% automatic and efficient solution for TLM
Problems and solutions for parallel execution of SystemC/TLM

(1) Execution order imposed by SystemC semantics
(2) Concurrent access to shared resources (e.g., $x++$ on a global variable)

$\Rightarrow$ No 100% automatic and efficient solution for TLM

Our proposal = additional constructs: Desynchronization (1) / Synchronization (2)
Approaches to parallelization

Efficient

Targets a wide subset of SystemC

Few/no modifications required
**SC-DURING: The Idea**

- **Unmodified SystemC**
- **Some computation delegated to other threads**
- **Weak synchronization between SystemC and threads thanks to tasks with duration**

- SC_THREAD_1
- SC_THREAD_2
- ... SC_THREAD_N
- OS thread_1
- OS thread_2
- OS thread_N

SystemC

OS thread
Simulated Time Vs Wall-Clock Time

Wall-clock time

Simulated time

Time elapse

Computation
Simulated Time in SystemC and sc-during

SystemC

A -> B

sc-during

P -> Q

// Computation
f();
wait(20);

Process A:
// Time taken by f
wait(20);

Process P:
g();
wait(20);
during(15, h);

Mathieu Moy (LIP)
Simulated Time in SystemC and SC-DURING

Process A:

```c
// Computation
f();
// Time taken by f
wait(20);
```

Process P:

```c
g();
wait(20);
during(15, h);
g();
i();
j();
```
Simulated Time in SystemC and SC-DURING

Process A:

```
// Computation
f();
// Time taken by f
wait(20);
```

Process P:

```
g();
wait(20);
```
Simulated Time in SystemC and SC-DURING

**Process A:**
// Computation
f();
// Time taken by f
wait(20);

**Process P:**
g();
wait(20);
during(15, h);
Simulated Time in SystemC and SC-DURING

**Process A:**

```cpp
// Computation
f();
// Time taken by f
wait(20);
```

**Process P:**

```cpp
g();
wait(20);
during(15, h);
```
Impact on Parallelism

P1  P2  P3  P4
Concurrency in an industrial platform

Number of SystemC threads active within a cycle (ST set-top-box case study):

- mpeg2 → h264
- h264
- mpeg2
- boot+init

<table>
<thead>
<tr>
<th>0 Proc.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4 and more</th>
</tr>
</thead>
</table>
Impact on Parallelism

P1

P2

P3

P4
Impact on Parallelism
Impact on Parallelism

Overlap between tasks $\rightsquigarrow$ parallel execution in $sc$-during
Execution of \texttt{during(T)}

SoCs and TLM

Matthieu Moy (LIP)
SC-DURING: First (Naive) Implementation

```cpp
void during(sc_core::sc_time d,
            std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}
```

Diagram:

```
A -----------------------
  
  B

  
  C

 Thread
```
SC-DURING: First (Naive) Implementation

```cpp
void during(sc_core::sc_time d,
            std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}
```

```cpp
during(d, f);
```

Thread creation

Thread execution

Wait for completion
void during(sc_core::sc_time d, std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}

during(d, f);
void during(sc_core::sc_time d, std::function<void()> f) {
    std::thread t(f);  // Thread creation
    sc_core::wait(d);  // SystemC executes
    t.join();  // Wait for completion
}

during(d, f);
void during(sc_core::sc_time d, 
    std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}

during(d, f);
void during(sc_core::sc_time d, 
           std::function<void()> f) {
    std::thread t(f); // Thread creation
    sc_core::wait(d); // SystemC executes
    t.join(); // Wait for completion
}
SC-DURING: First (Naive) Implementation

```cpp
void during(sc_core::sc_time d,
        std::function<void()> f) {
    std::thread t(f);  // Thread creation
    sc_core::wait(d);  // SystemC executes
    t.join();  // Wait for completion
}
```

1. **Thread creation**
2. **SystemC executes**
3. **Wait for completion**

During(d, f);
**SC-DURING: New Synchronization Primitives**

`extra_time(t):` Increase duration of current task

```
while (!c) {
    extra_time(10);
    catch_up();
    // Ensures fairness
}
```

```
x++;
// Forbidden in sc-during task
sc_call({ x++; });
// OK
```
**SC-DURING: New Synchronization Primitives**

**extra_time(t):** Increase duration of current task

P \[\text{wait}(5)\] initial duration extra time

**catch_up():** Wait for SystemC to reach the end of the task

\[
\text{while} \ (\neg c) \ \{ \\
\quad \text{extra_time}(10); \\
\quad \text{catch_up}(); \quad // \text{Ensures fairness} \\
\}\n\]
**SC-DURING: New Synchronization Primitives**

**extra_time(t):** Increase duration of current task

```
wait(5)
```

**catch_up():** Wait for SystemC to reach the end of the task

```
while (!c) {
    extra_time(10);
    catch_up(); // Ensures fairness
}
```

**sc_call(f):** Call function `f` in the context of SystemC

```
x++; // Forbidden in
    // sc-during task
sc_call([]){ x++; }; // OK
```
**SC-DURING: Implementations**

SystemC

- SC_THREAD_1
- SC_THREAD_2
- ... (tiles)
- SC_THREAD_N

OS Thread

- sync_task_1
- sync_task_2
- ... (tiles)
- sync_task_N

- OS thread_1
- OS thread_2
- ... (tiles)
- OS thread_N

Strategies:

- **SEQ**  Sequential (= reference)
- **THREAD**  Thread creation + destruction for each task
- **POOL**  Pre-allocated set of threads
- **ONDEMAND**  Thread created on demand and reused
SC-DURING: Results

Loose timing (explicit synchronization)

Fine-grained synchronization

Test machine: \(4 \times 12 = 48\) cores
Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```c
imgReady = true;  \textbf{while}(!imgReady)
wait(5, SC_US);  \textbf{while}(1, SC_US);
writeIMG();  \textbf{wait}(10, SC_US);
wait(10, SC_US);  \textbf{readIMG}();
```

⇒ bug never seen in simulation
Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```c
imgReady = true;
wait(5, SC_US);
writeIMG();
wait(10, SC_US);

||
while(!imgReady)
wait(1, SC_US);
wait(10, SC_US);
readIMG();
```

⇒ bug never seen in simulation

```c
during(15, SC_US, []{
    imgReady = true;
    writeIMG();
});
```

⇒ strictly more behaviors, including the buggy one
Model Faithfulness

- Extra behaviors of the model (A)
- Unmodeled behaviors (B)
- Exactly modeled behaviors (C)

Matthieu Moy (LIP)  Transaction-Level Models of SoCs  February 2018  <53/70>
New way to express concurrency in the platform

Allows parallel execution of loosely-timed systems

Exposes more bugs (⚠️ faithfulness Vs correction)

Next steps (skipped from this talk):

- Worker threads Vs platform partitioning: DistemC
- Exploit FIFO-based communication: FOFIFON
- Integration in the design-flow: HLS code wrapping
This section

4 Extra-Functional Properties in TLM
- Time and Parallelism
- Power and Temperature Estimation
Power and Temperature Estimation

An example

“How to validate embedded software that regulates the chip’s temperature?”

```c
while (true) {
    // Temperature of one or more locations of the chip
    read_sensors();

    compute();

    // Reduce frequency/voltage,
    // emergency stop, ...
    control_actuators();
}
```
Power and Temperature Estimation

What precision? What applications?

class control_actuators()

read_sensors()
Power and Temperature Estimation

What precision? What applications?

void usage (char *name) {
    printf("usage\n");
    printf("%s -a [-c file", name);
    exit (1);
    printf ("\n");
    sendf (print ("%s [" file]]")]");

control_actuators()}

read_sensors()
Power and Temperature Estimation
What precision? What applications?

control_actuators()

read_sensors()

Arbitrary Temperature
Power and Temperature Estimation

What precision? What applications?

```c
void usage (char *name)
{
    printf ("usage\n";\n    printf ("%s -o [-c file", name);\n    printf (" [\n file
    sendf (printf ("[\n file
);\n
control_actuators()
```

```c
void read_sensors()
```

Scenario
Power and Temperature Estimation

What precision? What applications?

`control_actuators()`

`read_sensors()`

Computation on a model
Power and Temperature Estimation

What precision? What applications?

control_actuators()

read_sensors()

Computation on a model
Power Consumption, Temperature, Heat Dissipation

Power (Joule effect)

Component

Dissipation (from another component)

Dissipation (to another component)

Dissipation (to environment)
Power Consumption, Temperature, Heat Dissipation

Power (Joule effect)

Component

Dissipation (from another component)

Dissipation (to another component)

Dissipation (to environment)

\[ \sim \text{differential equations, solved by dedicated solvers} \]
Estimation with Power-State Models

// SystemC Process
void compute() {
    while (true) {
        f();
        wait(10);
        wait();
    }
}
Estimation with Power-State Models

// SystemC Process
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10);
        set_state("idle");
        wait();
    }
}
From States to Consumption

State: sleep, run, idle, run

Consumption: 0 watt, 0.1 watt, 0.4 watt, 0.4 watt

Energy: (Consumption $\times$ Time) = 2.5 Joules
From Power to Temperature

State:
- sleep
- run
- idle
- run

Consumption:
- 0 watt
- 0.1 watt
- 0.4 watt
- 0.4 watt

Temperature:
- 20°C
- 40°C

Matthieu Moy (LIP)
**Traffic Models**

- **CPU**: `process = C++ code`
- **ITC**
- **VGA**
- **Timer**

**TLM Bus**

- **Data RAM**
- **Instruction RAM**
- **GPIO**

**Consumption**

- Consumption = \( f(\text{bits transmitted}) \)
- Consumption = \( f'(\text{bits processed}) \)
Traffic Model and Loosely Timed Models

Real System
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

\[ f(); \text{wait}(40); \quad g(); \text{wait}(35); \]
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Energy

Energy +3

Energy +6 total=9

f(); wait(40); g(); wait(35);

total=9

Temperature Unrealistic peaks Frequency 3 40 trans/sec 6 35 trans/sec

Mathieu Moy (LIP)
Traffic Model and Loosely Timed Models

Real System

\[
f() \text{; wait(40);} \text{; g() ; wait(35);}\
\]

Loosely-Timed Model

Energy

Temperature

Energy

Temperature

Total=9

+6

+3

Matthieu Moy (LIP)
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Energy

Temperature

Unrealistic peaks

$\text{total}=9$

$+3$

$+6$

$\text{total}=9$

$f(); \text{wait}(40);\ g(); \text{wait}(35);$
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

$f(); \text{wait}(40)$; $g(); \text{wait}(35)$;

$\frac{3}{40}$ trans/sec $\frac{6}{35}$ trans/sec
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

\[ \frac{3}{40} \text{ trans/sec} \quad \frac{6}{35} \text{ trans/sec} \]

Energy

Total = 9
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

Energy

Temperature

\[ \frac{3}{40} \text{ trans/sec} \quad \frac{6}{35} \text{ trans/sec} \]

Total = $9$

SoCs and TLM

Compilation

Verification

Extra-functional

Conclusion

Matthieu Moy (LIP)

Transaction-Level Models of SoCs

February 2018

< 62 / 70 >
Cosimulation SystemC and Extra-Functional Solver

Functionality can depend on extra-functional data (e.g.: temperature sensor)
Cosimulation of SystemC and Extra-Functional Solver

SystemC

Transaction-Level Models of SoCs

February 2018
Cosimulation of SystemC and Extra-Functional Solver

SystemC runs simulation up to end of instant $t_2$.
SystemC sends a request for extra-functional simulation on $[t, t + d]$.
Extra-functional solver does the computation on the interval $[t, t + d]$.
SystemC resumes simulation at the beginning of instant $t + d$.

Simulation Instant (Zero-time)
Cosimulation of SystemC and Extra-Functional Solver

SystemC - Function

Simulation Instant

$\text{Simulation Instant} \quad t = 0$

Simulation Interval

$\text{Simulation Interval} \quad t \in ]0, 3[ $

Simulation Instant

$\text{Simulation Instant} \quad t = 3$
Cosimulation of SystemC and Extra-Functional Solver

SystemC - Function

P/T^o - P/T^o - ...

Fonction
Cosimulation of SystemC and Extra-Functional Solver

1 SystemC runs simulation up to end of instant $t$
Cosimulation of SystemC and Extra-Functional Solver

1. SystemC runs simulation up to end of instant $t$
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$
Cosimulation of SystemC and Extra-Functional Solver

1. SystemC runs simulation up to end of instant $t$
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$
3. Extra-functional solver does the computation on the interval
Cosimulation of SystemC and Extra-Functional Solver

1. SystemC runs simulation up to end of instant $t$
2. SystemC sends a request for extra-functional simulation on $[t, t + d]$.
3. Extra-functional solver does the computation on the interval.
4. SystemC resumes simulation at beginning of instant $t + d$. 
Extra-Functional Events

1. SystemC runs simulation until end of instant \( t \)
Extra-Functional Events

1. SystemC runs simulation until end of instant $t$
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until “too hot”
Extra-Functional Events

1. SystemC runs simulation until end of instant $t$
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until “too hot”
3. Extra-functional runs simulation, encounters stop condition
Extra-Functional Events

1. SystemC runs simulation until end of instant $t$
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until “too hot”
3. Extra-functional runs simulation, encounters stop condition
4. SystemC resumes earlier than expected with interrupt.

Matthieu Moy (LIP)

Transaction-Level Models of SoCs

February 2018
Extra-Functional Events

1. SystemC runs simulation until end of instant $t$
2. SystemC requests a extra-functional simulation in $[t, t + d]$ or until “too hot”
3. Extra-functional runs simulation, encounters stop condition
4. SystemC resumes earlier than expected with interrupt.
Results

SoCs and TLM Compilation Verification

Extra-functional Results

Matthieu Moy (LIP)
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. Compilation of SystemC/TLM
3. Verification of SystemC/TLM
4. Extra-Functional Properties in TLM
5. Conclusion
Transaction-Level Models of Systems-on-a-Chip
Can they be Fast, Correct and Faithful?
Conclusion

- **Fast**
  - Optimized compiler
  - Parallelization techniques
  - High abstraction level (Loose Timing)

- **Correct**
  - Formal verification

- **Faithful**
  - More ways to express concurrency
  - Preserve Faithfulness of Temperature Models for Loose Timing
The new CASH Team, LIP (ENS-Lyon)

Compilation and Analysis for Software and Hardware

HPC Applications

Sequential Program

Polyhedral Model

Parallelism Extraction

Intermediate Parallel Representation

Dataflow Semantics

Simulation

Optimization

Analysis

Abstract Interpretation

Code Generation

Hardware (FPGA)

Software (CPU & accelerators)

Christophe Alias, Laure Gonnord, Matthieu Moy
Questions?
Sources

(Peter John Bishop, CC Attribution-Share Alike 3.0 Unported)

http://www.fotopedia.com/items/flickr-367843750
(oskay@fotopedia, CC Attribution 2.0 Generic)