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

Matthieu Moy
Verimag (Grenoble INP)
Grenoble, France

September 2012
Outline

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

Software

Hardware
Hardware/Software Design Flow

Traditional Design-Flow

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

Time
Hardware/Software Design Flow

Traditional Design-Flow

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

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
4. 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. Software Development
5. TLM Model
Hardware/Software Design Flow

Traditional Design-Flow

Specification, Algorithm

RTL Design

Synthesis

Factory

Software Development

Integration

Validation

Transaction-Level Model based

Specification, Algorithm

RTL Design

Synthesis

Factory

Integration

Validation

TLM Model

Software Development

Matthieu Moy (Verimag)
Hardware/Software Design Flow

Traditional Design-Flow

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

Transaction-Level Model based

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

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 ($\approx 1000x$ faster than RTL)
  - Lightweight modeling effort ($\approx 10x$ 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

10000 1000 100 10 1 x3 x60 x20
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development
Uses of Functional Models

Reference for Hardware Validation

虚拟原型用于软件开发
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

Non-functional

Conclusion

Matthieu Moy (Verimag)
Transaction-Level Models of SoCs
September 2012

< 11 / 70 >
Uses of Functional Models

Reference for Hardware Validation

Virtual Prototype for Software Development

Unmodified Software

Virtual Prototype for Software Development

SoCs and TLM  
Compiled  
Verification  
Non-functional  
Conclusion

Matthieu Moy (Verimag)  
Transaction-Level Models of SoCs  
September 2012 < 11 / 70 >
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 Non-Functional Models
Timing, Power consumption, Temperature Estimation
Use of Non-Functional Models
Timing, Power consumption, Temperature Estimation

Estimated ≈ Actual

Matthieu Moy (Verimag)
Use of Non-Functional Models

Timing, Power consumption, Temperature Estimation

Unmodified Power/Temperature-Aware Software

Estimated ≈ Actual

Matthieu Moy (Verimag)
Summary: Expected Properties of TLM Programs

SystemC/TLM Programs should

- Simulate fast,
- Satisfy correctness criterions,
- Reflect faithfully functional and non-functional properties of the actual system.
Outline

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

2. Compilation of SystemC/TLM

3. Verification of SystemC/TLM

4. Non-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
This section

Compilation of SystemC/TLM
- Front-end
- Optimization and Fast Simulation
SoCs and TLM

Compilation

Verification

Non-functional

Conclusion

SystemC Front-End

- In this talk: Front-end = “Compiler front-end” (AKA “Parser”)

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...
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://systemc.org/](http://systemc.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://systemc.org/](http://systemc.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)
   $\sim$ Write a C++ front-end or reuse one (g++, clang, EDG, ...)

2. Architecture built at runtime, with C++ code
   $\sim$ 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;
    }
}
```
Challenges and Solutions with SystemC Front-Ends

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

2. Architecture built at runtime, with C++ code
   - 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;
}
```
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 (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

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

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

Can we be here?
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)

LLVM bitcode

Execute elaboration

Identify SC constructs

%this not known
Cannot compute %port

%this is fixed

Architecture

Execute dependencies

Intermediate Representation

SystemC construct is still a normal function

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

...%port = expr1(%this)%data = expr2
...%port = expr1(%this)%data = expr2

...%port = expr1(%this)%data = expr2

SCWrite
- data = ??
- port = ??

...%port = expr1(%this)%data = expr2

SCWrite
- data = \{\begin{array}{l}
\text{Process 0} \rightarrow \text{data } d_0 \\
\text{Process 1} \rightarrow \text{data } d_1 \\
\end{array}\}
- port = \{\begin{array}{l}
\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

- **Typical Transaction Journey**
- **CPU**
- **Bus**
- **RAM**
- **T1**
- **T2**

**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**

Matthieu Moy (Verimag)
Transaction-Level Models of SoCs
September 2012
Typical Transaction Journey

CPU

... 
socket.write(addr, data);
...

Bus

0x6000
0x5000
0x4000
0x3000
0x2000
0x1000
0x0000

RAM

0x6000

T2

0x4000

T1

RAM

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

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

Matthieu Moy (Verimag)
Transaction-Level Models of SoCs
September 2012
Typical Transaction Journey

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

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

Call virtual method on socket

Forward method call to target socket

Another virtual method call

Forwarded to target socket

Ends-up calling target module’s method

Matthieu Moy (Verimag)
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
socket.write(0x5500, data);
```

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

```c
... socket.write(0x5500, data); ...
```

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

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

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

```c
socket.write(0x5500, data);
```

Get actual port addr from PinaVM

Follow path to bus

Address Decoding

Find target socket at this address

Find function in target module

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

```c
... socket.write(0x5500, data);
...

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

```c
... socket.write(0x5500, data);
...

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

- Possible optimizations:
  - Replace call to `socket.write()` with `RAM.write()`
  - Possibly inline it
Optimized Compilation for SystemC

C++ → Front end → IR → Code Generation → Link → Executable → Execution

Optimizer
Optimized Compilation for SystemC

SystemC

C++

Front end

IR

Optimizer

Code Generation

Link

Executable

Execution
Optimized Compilation for SystemC

- C++
- Front end
- IR
- Code Generation
- Link
- Executable
- JIT
- Execution
- SystemC
- Optimizer
Optimized Compilation for SystemC

SystemC

C++

Front end

IR

Code Generation

Link

Optimizer

JIT

Executable

Execution
Outline

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

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

SystemC
Concurrent program

Asynchronous automata
+ scheduler

Synchronous automata

$T_1 \times T_2 \times T_3 \times \text{Sch}$

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

$T_1 \bigotimes T_2 \bigotimes T_3$
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: $T_1 \times T_2 \times T_3 \times \text{Sch}$

SystemC Concurrent program

Asynchronous automata
Dedicated product
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
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
SystemC kernel: Direct semantics
Scheduler
Communication: Direct semantics
The SystemC scheduler

- Non-preemptive scheduler
- Non-deterministic processes election

Select process

Run → Init → Update → Time elapse

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

SystemC Concurrent program

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

- Asynchronous automata
  \[ T_1 \otimes T_2 \otimes T_3 \]

- Dedicated product
  \[ T_1 \otimes T_2 \otimes T_3 \]
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

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

Asynchronous automata

\[ T_1 \times T_2 \times T_3 \]

Asynchronous product

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

Asynchronous automata

\[ T_1 \times T_2 \times T_3 \]

Dedicated product
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

$T_1 \times T_2 \times T_3 \times \text{Sch}$

Asynchronous automata

$T_1 \otimes T_2 \otimes T_3$

Asynchronous automata

Dedicated product

SystemC Concurrent program
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

SystemC Concurrent program

\[ T_1 \odot T_2 \odot T_3 \]
Asynchronous automata

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

Dedicated product

Transaction-Level Models of SoCs
SystemC to Spin: encoding events

- notify/wait for event $E^k$:

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

- $W_p$: integer associated to process $p$.
  
  $W_p = k \Leftrightarrow$ “process $p$ 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 $p$
  $T_p = \text{next execution time for process } p$

```
$\text{p::wait}(d)$:
$T_p := T_p + d$
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 $p$
  $T_p = \text{next execution time for process } p$

$p::\text{wait}(d)$:

\[
T_p := T_p + d \\
\text{blocked}(T_p \leq \min_{i \in P} (T_i)) \\
W_i := 0
\]

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

$p::\text{wait}(E^k)$:

\[
W_p := K \\
\text{blocked}(W_p = 0)
\]

$p::\text{notify}(E^k)$:

\[
\forall i \in P \mid W_i = K \\
W_i := 0 \\
T_i := T_p
\]
SystemC to Spin: results

![Graph showing SystemC to Spin results](image-url)

- **PinaVM**
- **[SPIN 07]**

---

Matthieu Moy (Verimag)
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

Dedicated product

SystemC
Concurrent program

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

Asynchronous automata
Outline

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

Non-functional Properties in TLM

- Time and Concurrency
  - jTLM
  - Parallelization: jTLM and SC-DURING
- Power and Temperature Estimation
SystemC/TLM vs. “TLM Abstraction Level”
SystemC/TLM vs. “TLM Abstraction Level”

- SystemC
  - Cycle accurate
  - RTL
  - Gate level

- TLM
  - TLM 2.0
  - Coroutines
  - Function calls
  - Parallelism
  - Clocks

- Alternative to SystemC
SystemC/TLM vs. “TLM Abstraction Level”

SystemC
- Cycle accurate
- RTL
- Clocks
- Coroutine semantics
- $\delta$-cycle

TLM
- Parallelism
- Function calls
SystemC/TLM vs. “TLM Abstraction Level”

- SystemC:
  - Cycle accurate
  - Clocks
  - RTL
  - Coroutine semantics
  - Gate level
  - $\delta$-cycle

- TLM:
  - Parallelism
  - Function calls
  - ?
SystemC/TLM vs. "TLM Abstraction Level"
jTLM: goals and peculiarities

- jTLM’s initial goal: define “TLM” independently of SystemC
  - Not cooperative (true parallelism)
  - Not C++ (Java)
  - No $\delta$-cycle

- Interesting features
  - Small and simple code ($\approx 500$ LOC)
  - Nice experimentation platform

- Not meant for production
Simulated Time Vs Wall-Clock Time

Wall-clock time

0 10 20 30 40

Simulated time

Time elapse

Computation
(Simulated) Time in SystemC and jTLM

SystemC

A  B

\[ \text{Process A:} \quad \text{// computation} \quad f(); \quad \text{// time taken by f} \quad \text{wait}(20, \text{SC_NS}); \]

\[ \text{Process P:} \quad \text{g();} \quad \text{awaitTime}(20); \quad \text{consumesTime}(15) \{ \text{h();} \} \]

jTLM

P  Q
(Simulated) Time in SystemC and jTLM

**Process A:**
```plaintext
// computation
f();
// time taken by f
wait(20, SC_NS);
```

**SystemC**

A -> B

**jTLM**

P -> Q
(Simulated) Time in SystemC and jTLM

Process A:

```cpp
// computation
f();
// time taken by f
wait(20, SC_NS);
```

```
Process P:
g();
awaitTime(20);
consumesTime(15) {
h();
}
```

```
Process Q:
i();
```

SystemC:

```
Process A:
f();
wait(20)
```

```
Process B:

```
```

jTLM:

```
Process P:
g();
awaitTime(20)
```

```
Process Q:
i();
```

```
```
(Simulated) Time in SystemC and jTLM

**Process A:**
```
// computation
f();
// time taken by f
wait(20, SC_NS);
```

**Process P:**
```
g();
awaitTime(20);
```
(Simulated) Time in SystemC and jTLM

**Process A:**
```cpp
// computation
f();
// time taken by f
wait(20, SC_NS);
```

**Process P:**
```cpp
g();
awaitTime(20);
consumesTime(15) {
    h();
}
```
(Simulated) Time in SystemC and jTLM

**Process A:**
```
// computation
f();
// time taken by f
wait(20, SC_NS);
```

**Process P:**
```
g();
awaitTime(20);
consumesTime(15) {
  h();
}
```
**Time à la SystemC**: `awaitTime(T)`

- By default, time does not elapse ⇒ instantaneous tasks
- `awaitTime(T)`:
  suspend and let other processes execute for $T$ time units

```c
f(); // instantaneous
awaitTime(20);
```
Task with Known Duration: \texttt{consumesTime}(T)

- **Semantics:**
  - Start and end dates known
  - Actions contained in task spread in between

- **Advantages:**
  - Model closer to actual system
  - Less bugs hidden
  - Better parallelization

```plaintext
consumesTime(15) {
  f1();
  f2();
  f3();
}

consumesTime(10) {
  g();
}
```
Addressing the Faithfulness Issue: Exposing Bugs

Example bug: mis-placed synchronization:

```java
imgReady = true; awaitTime(5); writeIMG();
awaitTime(10); while(!imgReady) awaitTime(1);
awaitTime(10); readIMG();
```

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

Example bug: mis-placed synchronization:

```
imgReady = true;       while(!imgReady)
awaitTime(5);         ||  awaitTime(1);
writeIMG();           awaitTime(10);
awaitTime(10);        readIMG();
```

⇒ bug never seen in simulation

```
consumesTime(15) {
  imgReady = true;       while(!imgReady)
  awaitTime(5);         ||  awaitTime(1);
  writeIMG();           awaitTime(10);
  }
```

⇒ strictly more behaviors, including the buggy one
Time Queue and `awaitTime(T)`

**Current instant**

**P, Q, R**

**Process P:**
- `f();`
- `awaitTime(50);`

**Process Q:**
- `h();`
- `awaitTime(30);`
- `g();`
- `awaitTime(30);`

**Process R:**
- `i();`
- `awaitTime(90);`
Time Queue and `awaitTime(T)`

Process P:
- `f();`
- `awaitTime(50);`

Process Q:
- `h();`
- `awaitTime(30);`
- `g();`
- `awaitTime(30);`

Process R:
- `i();`
- `awaitTime(90);`
Time Queue and \texttt{awaitTime}(T)

Current instant

\texttt{awaitTime}(30)

**Process P:**
\begin{align*}
f() ; \\
\triangleright \texttt{awaitTime}(50) ;
\end{align*}

**Process Q:**
\begin{align*}
h() ; \\
\triangleright \texttt{awaitTime}(30) ; \\
g() ; \\
\texttt{awaitTime}(30) ;
\end{align*}

**Process R:**
\begin{align*}
\triangleright \texttt{i}() ; \\
\texttt{awaitTime}(90) ;
\end{align*}
Time Queue and `awaitTime(T)`

Current instant

**Process P:**
\[ f(); \]
\[ \triangleright awaitTime(50); \]

**Process Q:**
\[ h(); \]
\[ \triangleright awaitTime(30); \]
\[ g(); \]
\[ \triangleright awaitTime(30); \]

**Process R:**
\[ i(); \]
\[ \triangleright awaitTime(90); \]
Time Queue and \texttt{waitForTime}(T)

Current instant

Process P:
\begin{verbatim}
f();
\downarrow \texttt{waitForTime}(50);
\end{verbatim}

Process Q:
\begin{verbatim}
h();
\downarrow \texttt{waitForTime}(30);
g();
\texttt{waitForTime}(30);
\end{verbatim}

Process R:
\begin{verbatim}
i();
\downarrow \texttt{waitForTime}(90);
\end{verbatim}
**Time Queue and** \texttt{awaitTime(T)}

**Current instant**

**Process P:**
\begin{align*}
f() & \\
\triangleright & \text{\texttt{awaitTime(50)}};
\end{align*}

**Process Q:**
\begin{align*}
h() & \\
\text{\texttt{awaitTime(30)}} & \\
\triangleright & \text{\texttt{g()}}; \\
\text{\texttt{awaitTime(30)}} & \\
\end{align*}

**Process R:**
\begin{align*}
i() & \\
\triangleright & \text{\texttt{awaitTime(90)}};
\end{align*}
Time Queue and \texttt{consumesTime(T)}

What about \texttt{consumesTime(T)}?
Time Queue and `consumesTime(T)`

Process P:
- `f();`  
- `consumesTime(50){`  
  - `g();`  
- `}`  
- `h();`  

Process Q:
- `i();`  
- `awaitTime(30);`  
- `j();`  
- `consumesTime(30){`  
  - `k();`  
- `}`

Process R:
- `l();`  
- `awaitTime(90);`
Time Queue and \texttt{consumesTime(T)}

Current instant

\texttt{consumesTime(50)}

\begin{itemize}
  \item Process \texttt{P}:
    \begin{verbatim}
    f();
    \texttt{consumesTime(50)}{
    g();
    }
    h();
    \end{verbatim}
  \item Process \texttt{Q}:
    \begin{verbatim}
    i();
    \texttt{awaitTime(30)};
    j();
    \texttt{consumesTime(30)}{
    k();
    }
    \end{verbatim}
  \item Process \texttt{R}:
    \begin{verbatim}
    l();
    \texttt{awaitTime(90)};
    \end{verbatim}
\end{itemize}
Time Queue and `consumesTime(T)`

**Process P:**

```plaintext
f();
consumesTime(50){
  g();
}
h();
```

**Process Q:**

```plaintext
i();
awaitTime(30);
j();
consumesTime(30){
k();
}
```

**Process R:**

```plaintext
l();
awaitTime(90);
```
Current instant

\text{awaitTime}(30)

Process P:
\begin{verbatim}
f();
\text{consumesTime}(50)\{
\quad g();
\}
\text{h();}
\end{verbatim}

Process Q:
\begin{verbatim}
i();
\text{awaitTime}(30);
\quad j();
\text{consumesTime}(30)\{
\quad k();
\}
\end{verbatim}

Process R:
\begin{verbatim}
\text{\textgreater{}l();}
\text{awaitTime}(90);
\end{verbatim}
Time Queue and `consumesTime(T)`

Process P:
- `f();`
- `consumesTime(50){
  g();
}
- `h();`  

Process Q:
- `i();`
- `awaitTime(30);`
- `j();`
- `consumesTime(30){
  k();
}

Process R:
- `l();`
- `awaitTime(90);`
Process P:
  f();
  consumesTime(50){
    g();
  }
  h();

Process Q:
  i();
  awaitTime(30);
  j();
  consumesTime(30){
    k();
  }

Process R:
  l();
  awaitTime(90);
Time Queue and `consumesTime(T)`

**Process P:**
```plaintext
f();
consumesTime(50) {
  g();
}
h();
```

**Process Q:**
```plaintext
i();
awaitTime(30);
  j();
  consumesTime(30) {
    k();
  }
```

**Process R:**
```plaintext
l();
  ▶ awaitTime(90);
**Time Queue and `consumesTime(T)`**

**Current instant `consumesTime(30)`**

**Process P:**
- `f();`  
- `consumesTime(50){`  
  - `g();`  
  - `h();`  
- `};`  

**Process Q:**
- `i();`  
- `awaitTime(30);`  
- `j();`  
- `k();`  
- `};`  

**Process R:**
- `l();`  
- `awaitTime(90);`
**Time Queue and** \texttt{consumesTime(T)}

**Current instant**

**Process P:**
\begin{verbatim}
f();
consumesTime(50){
    g();
}
h();
\end{verbatim}

**Process Q:**
\begin{verbatim}
i();
awaitTime(30);
j();
consumesTime(30){
    k();
}
\end{verbatim}

**Process R:**
\begin{verbatim}
l();
\end{verbatim}
\begin{verbatim}
\triangleright awaitTime(90);
\end{verbatim}
**Time Queue and** `consumesTime(T)`

Current instant

Time Elapse

Process P:
```c
f();
consumesTime(50){
g();
} 
} 

h();
```

Process Q:
```c
i();
awaitTime(30);
j();
consumesTime(30){
  k();
}
```

Process R:
```c
l();
▷ awaitTime(90);
```
Time Queue and \texttt{consumesTime(T)}

Process P:
\begin{verbatim}
f();
consumesTime(50){
g();
}
\textcolor{red}{\triangleright}h();
\end{verbatim}

Process Q:
\begin{verbatim}
i();
awaitTime(30);
j();
consumesTime(30){
\textcolor{red}{\triangleright}k();
}
\end{verbatim}

Process R:
\begin{verbatim}
l();
\textcolor{red}{\triangleright}awaitTime(90);
\end{verbatim}
Parallelization

jTLM’s Semantics

Simultaneous tasks run in parallel
Parallelization

jTLM’s Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don’t

Parallelizing within $\delta$-cycle = great if you have clocks
Simulated time is the bottleneck with quantitative/fuzzy time
Can we apply the idea of duration to SystemC?
Parallelization

**jTLM’s Semantics**

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don’t
- Overlapping tasks do

Parallelization within δ-cycle = great if you have clocks

Simulated time is the bottleneck with quantitative/fuzzy time

Can we apply the idea of duration to SystemC?
Parallelization

jTLM’s Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don’t
- Overlapping tasks do

Back to SystemC:
- Parallelizing within $\delta$-cycle = great if you have clocks
- Simulated time is the bottleneck with quantitative/fuzzy time
Parallelization

jTLM’s Semantics

- Simultaneous tasks run in parallel
- Non-simultaneous tasks don’t
- Overlapping tasks do

Back to SystemC:
- Parallelizing within $\delta$-cycle = great if you have clocks
- Simulated time is the bottleneck with quantitative/fuzzy time

Can we apply the idea of duration to SystemC?
SC-DURING: the Idea

- Goal: allow during tasks in SystemC
  - Without modifying SystemC
  - Allowing physical parallelism

- Idea: let SystemC processes delegate computation to a separate thread
SC-DURING: Sketch of Implementation

```cpp
void during(sc_core::sc_time duration,
    boost::function<void()> routine) {
  ➊ boost::thread t(routine); // create thread
  ➋ sc_core::wait(time); // let SystemC execute
  ➋ t.join(); // wait for thread completion
}
```

Matthieu Moy (Verimag)
Transaction-Level Models of SoCs
September 2012
void during(sc_core::sc_time duration,
boost::function<void()> routine) {
① boost::thread t(routine); // create thread
② sc_core::wait(time); // let SystemC execute
③ t.join(); // wait for thread completion
}

during(5, f);

pthread

SC-DURING: Sketch of Implementation
void during(sc_core::sc_time duration,
    boost::function<void()> routine) {
  u0 boost::thread t(routine); // create thread
  u1 sc_core::wait(time); // let SystemC execute
  u2 t.join(); // wait for thread completion
}

during(5, f);
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
  boost::thread t(routine); // create thread
  sc_core::wait(time); // let SystemC execute
  t.join(); // wait for thread completion
}

during(5, f);
**SC-DURING: Sketch of Implementation**

```cpp
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
    boost::thread t(routine); // create thread
    sc_core::wait(time); // let SystemC execute
    t.join(); // wait for thread completion
}
```

```cpp
during(5, f);
```

**Diagram:**

- **A**
- **B**
- **C**
- **create thread**
- **wait(d)**
- **routine**
**SC-DURING: Sketch of Implementation**

```cpp
void during(sc_core::sc_time duration,
            boost::function<void()> routine) {
    boost::thread t(routine); // create thread
    sc_core::wait(time); // let SystemC execute
    t.join(); // wait for thread completion
}
```

during(5, f);
void during(sc_core::sc_time duration,
    boost::function<void()> routine) {

    boost::thread t(routine); // create thread
    sc_core::wait(time); // let SystemC execute
    t.join(); // wait for thread completion
}

during(5, f);
**SC-DURING: Synchronization**

`extra_time(t):` increase current task duration

```
wait(5)
```

![Diagram showing synchronization in SC-DURING with `extra_time` function and `wait` function.](image)

---

Matthieu Moy (Verimag)

Transaction-Level Models of SoCs

September 2012
**SC-DURING: Synchronization**

`extra_time(t)`: increase current task duration

`catch_up(t)`: block task until SystemC’s time reaches the end of the current task

```c
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```
**SC-DURING: Synchronization**

**extra_time(t):** increase current task duration

```
wait(5)
```

**catch_up(t):** block task until SystemC’s time reaches the end of the current task

```
while (!c) {
    extra_time(10, SC_NS);
    catch_up(); // ensures fairness
}
```

**sc_call(f):** call function \( f \) in the context of SystemC

```
e.notify(); // Forbidden in during tasks
```

```
sccall(e.notify()); // OK (modulo syntax)
```
SC-DURING: Actual Implementation

SystemC

SC_THREAD_1 → sync_task_1 → OS thread_1
SC_THREAD_2 → sync_task_2 → OS thread_2
∀: SC_THREAD_N → sync_task_N → OS thread_N

Strategies:
- SEQ  Sequential (= reference)
- THREAD Thread created/destroyed each time
- POOL Pre-allocated thread pool
- ONDEMAND Thread created on demand and reused later
SC-DURING: Results

Test machine has $4 \times 12 = 48$ cores

Loosely-Timed Models

Fine-grain Timing

thread, during_quant  
thread, during_sync
ondemand, during_quant
ondemand, during_sync

Number of CPU in the platform
SC-DURING and jTLM: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed systems
- Exposes more bugs (⚠️ faithfulness Vs correction)
This section

Non-functional Properties in TLM
- Time and Concurrency
  - jTLM
  - Parallelization: jTLM and SC-DURING
- Power and Temperature Estimation
Power estimation in TLM: Power-state Model

// SystemC thread
void compute() {
    while (true) {
        f();
        wait(10, SC_MS);
        wait(irq);
    }
}
Power estimation in TLM: Power-state Model

Consumption depends on:
- Activity state (switching activity inside component)
- Electrical state (voltage, frequency)

```c
// SystemC thread
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10, SC_MS);
        set_state("idle");
        wait(irq);
    }
}
```
Power estimation in TLM: Power-state Model

// SystemC thread
void compute() {
    while (true) {
        set_state("run");
        f();
        wait(10, SC_MS);
        set_state("idle");
        wait(irq);
    }
}

Consumption depends on:
- Activity state (switching activity inside component)
- Electrical state (voltage, frequency)
- Traffic (stimulation by other components)
Traffic Models

CPU

process = C++ code

ITC

VGA

Timer

Data RAM

Instruction RAM

GPIO

TLM Bus

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

Loosely-Timed Model:
Traffic Model and Loosely Timed Models

Real System

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

Loosely-Timed Model

Energy

+3

\[ \text{total}=9 \]

+6

\[ \text{total}=9 \]
Traffic Model and Loosely Timed Models

Real System

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

Loosely-Timed Model

Energy

+3

+6 total=9

Temperature
Traffic Model and Loosely Timed Models

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

Loosely-Timed Model

Energy
+3
+6

Temperature

Unrealistic peaks

total=9
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

\[ \frac{3}{40} \text{ trans/sec} \quad \frac{6}{35} \text{ trans/sec} \]
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

Energy

SoCs and TLM
Compilation Verification Non-functional Conclusion

Matthew Moy (Verimag) Transaction-Level Models of SoCs September 2012 < 65 / 70
Traffic Model and Loosely Timed Models

Real System

Loosely-Timed Model

Frequency

Energy

Temperature

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

$\frac{3}{40}$ trans/sec

$\frac{6}{35}$ trans/sec

total=9
SystemC and Temperature Solver Cosimulation

Functionality can depend on non-functional data (e.g. validate power-management policy)
<table>
<thead>
<tr>
<th>SoCs and TLM</th>
<th>Compilation</th>
<th>Verification</th>
<th>Non-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. **Non-functional Properties in TLM**
5. **Conclusion**
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
Conclusion

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

- **Correct**
  - Formal verification
  - *Runtime Verification*

- **Faithful**
  - More ways to express concurrency
  - Preserve Faithfulness of Temperature Models for Loose Timing
  - *Semantics for timed systems*
  - *Refinement techniques from functional to timed models*
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)