Modeling of Time in Discrete-Event Simulation for Systems-on-Chips

Matthieu Moy

Verimag (Grenoble INP)
Grenoble, France

November 2012
Modern Systems-on-a-Chip
Modern Systems-on-a-Chip

Software

Hardware
Hardware/Software Design Flow

Traditional Design-Flow

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

Time

Transaction-Level Model based Specification, Algorithm
RTL Design
Synthesis
TLM Model
Integration
Validation
Hardware/Software Design Flow

Traditional Design-Flow

Time

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

\[ \text{cost} > 1,000,000 \, \text{\$!} \]
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

Time in Discrete-Event Simulation
November 2012
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

Time in Discrete-Event Simulation
November 2012
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

Time in Discrete-Event Simulation

November 2012
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

Matthieu Moy (Verimag)
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

Time in Discrete-Event Simulation
November 2012
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. jTLM: Experiments Without SystemC
3. Back to SystemC: sc-during
4. Conclusion
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. jTLM: Experiments Without SystemC
3. Back to SystemC: sc-during
4. Conclusion
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

- **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
  - process = C++ code
- ITC
- VGA
- Timer
- Data RAM
- Instruction RAM
- GPIO

TLM Bus
Performance of TLM

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

Simulation time (second) logarithmic scale

x20

x60

x3
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

SoCs and TLM

jTLM

sc-during

Conclusion

Matthieu Moy (Verimag)

Time in Discrete-Event Simulation

November 2012

< 10 / 42 >
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
Uses of Functional Models

Reference for Hardware Validation

Unmodified Software

Virtual Prototype for Software Development

SoCs and TLM
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
SystemC

- SystemC is ...
  - a library for C++
  - a discrete-event simulator
  - well-suited for TLM
  - (an IEEE standard)

- SystemC/TLM programs are ...
  - fast (details abstracted away, efficiency of C++)
  - not fast enough (no physical parallelism)
  - too deterministic?
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. jTLM: Experiments Without SystemC
3. Back to SystemC: sc-during
4. Conclusion
SystemC/TLM vs. “TLM Abstraction Level”
SystemC/TLM vs. "TLM Abstraction Level"
SystemC/TLM vs. “TLM Abstraction Level”

SystemC
- Cycle accurate
- Clocks
- RTL
- Coroutine semantics
- Gate level

TLM
- Parallelism
- Function calls
- δ-cycle

Alternative to SystemC

Matthieu Moy (Verimag)
SystemC/TLM vs. “TLM Abstraction Level”

SystemC

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

TLM

- Parallelism
- Function calls
- TLM 2.0

jTLM is an Alternative to SystemC

Matthieu Moy (Verimag)
SystemC/TLM vs. “TLM Abstraction Level”

SystemC
- Cycle accurate
- RTL
- Clocks
- Coroutine semantics
- Gate level

TLM
- Parallelism
- Function calls
- jTLM = Alternative to SystemC

jTLM
jTLM: Goals and Peculiarities

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

- Interesting features
  - Small and simple code (≈ 500 LOC)
  - Nice experimentation platform

- Not meant for production
Simulated Time Vs Wall-Clock Time
(Simulated) Time in SystemC and jTLM

SystemC

A \rightarrow B

jTLM

P \rightarrow Q
(Simulated) Time in SystemC and jTLM

Process A:

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

In SystemC:
- **A**
- **B**

In jTLM:
- **P**
- **Q**
(Simulated) Time in SystemC and jTLM

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

---

SystemC

P

Q

jTLM
(Simulated) Time in SystemC and jTLM

SystemC

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

jTLM

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

**Process A:**

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

**Process P:**

```javascript
// awaitTime

```
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: \( \text{consumesTime}(T) \)

**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();
}
```
Execution of $\text{consumesTime}(T)$

Slow computation:
- Simulated time blocked
- Task starts
- Task finishes
- Wall-clock time

Fast computation:
- Computation ends
- Task finishes
- Rest of the platform drives time
- Wall-clock time
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:

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

⇒ bug never seen in simulation

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

⇒ strictly more behaviors, including the buggy one
```
Parallelization

P1 →
P2 →
P3 →
P4 →

jTLM’s Semantics

- Simultaneous tasks run in parallel

Simulated time is the bottleneck with quantitative/fuzzy time.
Can we apply the idea of duration to SystemC?
(Answer in next section)
Parallelization

jTLM’s Semantics

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

jTLM’s Semantics

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

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

jTLM’s Semantics
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? (Answer in next section)
Time Queue and \texttt{awaitTime(T)}

**Process P:**
- \texttt{f();}
- \texttt{awaitTime(50);}

**Process Q:**
- \texttt{h();}
- \texttt{awaitTime(30);}
- \texttt{g();}
- \texttt{awaitTime(30);}

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

Current instant

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

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

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

**Current instant**

- `awaitTime(30)`

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

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

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

Current instant

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)

Process P:
\begin{itemize}
  \item \texttt{f();}
  \item $\triangleright$ \texttt{awaitTime(50);}
\end{itemize}

Process Q:
\begin{itemize}
  \item \texttt{h();}
  \item \texttt{awaitTime(30);}
  \item $\triangleright$ \texttt{g();}
  \item \texttt{awaitTime(30);}
\end{itemize}

Process R:
\begin{itemize}
  \item \texttt{i();}
  \item $\triangleright$ \texttt{awaitTime(90);}
\end{itemize}
Time Queue and \texttt{consumesTime(T)}

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

---

**Process P:**

\begin{verbatim}
\begin{align*}
\texttt{f()} &;
\texttt{consumesTime(50)} \{ \\
    \texttt{g()} &;
\}
\texttt{h();}
\end{align*}
\end{verbatim}

**Process Q:**

\begin{verbatim}
\begin{align*}
\texttt{i()} &; \\
\texttt{awaitTime(30)} &; \\
\texttt{j()} &; \\
\texttt{consumesTime(30)} &\{ \\
    \texttt{k()} &; \\
\}
\end{align*}
\end{verbatim}

**Process R:**

\begin{verbatim}
\begin{align*}
\texttt{l()} &; \\
\texttt{awaitTime(90)} &;
\end{align*}
\end{verbatim}
Time Queue and \texttt{consumesTime(T)}

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

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

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

**Process P:**

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

**Process Q:**

```plaintext
i();
awaitTime(30);
j();
consumesTime(30){
  k();
} avoidTime(90);
```

**Process R:**

```plaintext
l();
awaitTime(90);
```
Time Queue and `consumesTime(T)`

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

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

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

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

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

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

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

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

**Process R:**
\begin{verbatim}
    l();
    \texttt{awaitTime(90)};
\end{verbatim}

Matthieu Moy (Verimag)  
Time in Discrete-Event Simulation  
November 2012
Time Queue and \texttt{consumesTime(T)}

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

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

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

Current instant: `consumesTime(30)`

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

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

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

\begin{itemize}
  \item \textbf{Process P:}
    \begin{verbatim}
    f();
    consumesTime(50){
      g();
    }
    h();
    \end{verbatim}
  \item \textbf{Process Q:}
    \begin{verbatim}
    i();
    awaitTime(30);
    j();
    consumesTime(30){
      k();
    }
    \end{verbatim}
  \item \textbf{Process R:}
    \begin{verbatim}
    l();
    ▶ 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
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

---

Matthieu Moy (Verimag)
Outline

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. jTLM: Experiments Without SystemC
3. Back to SystemC: sc-during
4. Conclusion
jTLM is cool ... 

... but nobody will use it.
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
void during(sc_core::sc_time duration,
boost::function<void()> routine) {

boost::thread t(routine); // create thread
sc_core::wait(duration); // let SystemC execute

// wait for thread completion

pthread

A
B
C
**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(duration); // let SystemC execute
    t.join(); // wait for thread completion
}
```

A

B

C

pthread

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

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

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

during(5, f);
```

Legend:
1. create thread
2. wait(d)
3. wait for thread completion
void during(sc_core::sc_time duration,
    boost::function<void()> routine) {
    boost::thread t(routine); // create thread
    sc_core::wait(duration); // let SystemC execute
    t.join(); // wait for thread completion
}

during(5, f);
void during(sc_core::sc_time duration,
        boost::function<void()> routine) {
    ① boost::thread t(routine); // create thread
    ② sc_core::wait(duration); // let SystemC execute
    ③ t.join(); // wait for thread completion
}
during(5, f);

Matthieu Moy (Verimag)  Time in Discrete-Event Simulation  November 2012  < 29 / 42 >
Wait ... are you saying that parallelization is just about fork/join?
Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...
Before

```
compute_in_systemc();

// my profiler says it’s performance critical.
// does not communicate with other processes.
big_computation();
wait(10, SC_MS);

next_computation();
```

After

```
compute_in_systemc();

// Won’t be a performance bottleneck anymore
during(10, SC_MS, big_computation);

next_computation();
```
Wait ... are you saying that parallelization is just about fork/join?

Well, sometimes it is ...
Wait ... are you saying that parallelization is just about fork/join? 

Well, sometimes it is ... 

... and sometimes it isn’t: 

**Time synchronization**: make sure things are executed at the right simulated time 

**Data/scheduler synchronization**: avoid data-race between tasks, processes and the SystemC scheduler.
extra_time(t): increase current task duration

```
wait(5)
```

\[
\text{while} (!c) \\
\text{extra_time}(10, \text{SC_NS}); \\
\text{catch_up();}
\]

SC-DURING: Synchronization
**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
}
```
EXTRA_TIME(): Sketch of Implementation

```cpp
void during(duration, routine) {
    end = now() + duration;
    boost::thread t(routine);
    // used to be just sc_core::wait(duration)
    while (now() != end) {
        sc_core::wait(end - now());
    }
    t.join();
}

void extra_time(duration) {
    end += duration;
}

void catch_up() {
    while (now() != end) {
        // avoid busy-waiting
        condition.wait();
    }
}
```
Temporal decoupling and \textit{sc--during}

Plain SystemC
\begin{verbatim}
f();
t_local += 42;
g();
t_local += 12;
wait(t_local);
t_local = 0;
i();
\end{verbatim}

\textit{sc--during}
\begin{verbatim}
f();
extra_time(42);
g();
extra_time(12);
catch_up();
i();
\end{verbatim}
**sc_call**: be cooperative for a while

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

- `e.notify();` // Forbidden in during tasks
- `sc_call("e.notify()");` // OK (modulo syntax)
- `sc_call("i++");` // implicit big lock, // no data-race
SC_CALL: Sketch of Implementation

during(5, f);

A
B
C

pthread
**SC_CALL: Sketch of Implementation**

during(5, f);

create thread

wait(d or sync_ev)
**SC_CALL: Sketch of Implementation**

during(5, f);

create thread

wait(d or sync_ev)

pthread
**SC_CALL: Sketch of Implementation**

```c
pthread during(5, f);
create thread
wait(d or sync_ev)
sc_call(g)
```
SC_CALL: Sketch of Implementation

during(5, f); create thread pthread

wait(d or sync_ev) notify sync_ev

sc_call(g)
**SC_CALL: Sketch of Implementation**

- `during(5, f);`
- `create thread`
- `wait(d or sync_ev)`
- `notify`
- `sync_ev`
- `sc_call_f = 0`
- `sc_call(g)`
SC_CALL: Sketch of Implementation

during(5, f);

create thread

wait(d or sync_ev)

notify sync_ev

wait

sc_call_f = 0

sc_call(g)
SC_CALL: Sketch of Implementation

during(5, f);

create thread

wait(d or sync_ev)

notify sync_ev

sc_call(g)

wait

sc_call_f = 0

join thread
void during(duration, f) {
    end = now() + duration;
    boost::thread t(f);
    while (now() != end) {
        // wait sync_ev
        // with timeout:
        sc_core::wait
            (sync_ev,
             end - now());
        if (sc_call_f) {
            sc_call_f();
            sc_call_f = 0;
            condition.notify();
        }
    }
    t.join();
}

void sc_call(f) {
    sc_call_f = f;
    // Implemented
    // with SystemC 2.3’s
    // async_request_update()
    async_notify_event
        (sync_ev);
    while(sc_call_f != 0) {
        condition.wait();
    }
}
**SC-DURING: Actual Implementation**

Possible strategies:

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

![Graph showing speedup vs number of CPUs]

Loosely-Timed Models

Fine-grain Timing

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

1. Introduction: Systems-on-a-Chip, Transaction-Level Modeling
2. jTLM: Experiments Without SystemC
3. Back to SystemC: sc-during
4. Conclusion
New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- jTLM: experimentation platform, new scheduler
- sc-during: jTLM’s ideas, implemented on top of SystemC. Still room for performance optimizations.

Try it:
https://forge.imag.fr/projects/sc-during/
jTLM and SC-DURING: Conclusion

- New way to express concurrency in the platform
- Allows parallel execution of loosely-timed (clockless) systems
- jTLM: experimentation platform, new scheduler
- sc-during: jTLM’s ideas, implemented on top of SystemC. Still room for performance optimizations.

Try it:
https://forge.imag.fr/projects/sc-during/

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)