Applying Formal Verification for better Dynamic Verification of SystemC Models

Nicolas Blanc, Daniel Kroening

To appear at ICCAD 2008

Oxford University Computing Laboratory
Outline

1. Motivation
2. Partial-Order Reduction
3. Scoot
4. Experimental Results
High-Level Models

- Emergence of system design languages
- HardwareC, SpecC, Handel-C, and SystemC
- Based on C / C++
- Allow joint modeling of both hardware and software components of a system
- Support for bit vectors, concurrency, synchronization, exception handling
SystemC

- Based on C++
  - No language extensions, but macros + library
  - Simulation using regular C++ compiler
SystemC

- Based on C++
  - No language extensions, but macros + library
  - Simulation using regular C++ compiler

SystemC Model

SystemC Library
SystemC

- Based on C++
  - No language extensions, but macros + library
  - Simulation using regular C++ compiler

SystemC Model → g++ → Executable

SystemC Library
SystemC

- Based on C++
  - No language extensions, but macros + library
  - Simulation using regular C++ compiler

SystemC Model

SystemC Library

```
g++ + Stimulus = Traces
```
SystemC

- Originally for fast, low-level circuit simulations
  - Verilog-like multi-valued logic (0, 1, X, Z)
  - Multiple drivers for a single signal

- Also offers
  - Bit-vector types
  - Fixed-point arithmetic
  - Concurrency

- Parts of SystemC are synthesizable
SC_MODULE(m) {
    sc_in<bool> data_in; // input port
    sc_in<bool> clock; // input port
    sc_out<bool> data_out; // output port

    void thread1() {
        void thread2();
    }

    int i;

    SCCTOR(m) { // Constructor
        SC_THREAD(thread1); sensitive << data_
        SC_THREAD(thread2); sensitive pos << c
    }
}
SystemC

SC_MODULE(m) {
    sc_in<bool> data_in; // input port
    sc_in<bool> clock; // input port
    sc_out<bool> data_out; // output port

    void thread1();
    void thread2();

    int i;

    SC_CTOR(m) { // Constructor
        SC_THREAD(thread1); sensitive << data_
        SC_THREAD(thread2); sensitive pos << c
SystemC

C/C++

Verilog/VHDL

Convenient modeling of both hardware and software
Applications of SystemC

Possible applications:

- Hardware model for co-simulation of embedded software
- Synthesis of algorithms
- Can serve as high-level model for hardware, in particular at the transaction level
Concurrency in SystemC

- Asynchronous interleaving semantics
  → Thread schedule is non-deterministic
Concurrency in SystemC

- Asynchronous interleaving semantics
  - Thread schedule is non-deterministic

- But: Interleaving only at specific locations
  - `wait()`
  - End of thread
  - No issues with atomicity
  - Does not really map onto usual pthread model

- Makes synthesis and model checking much easier!
Concurrency in SystemC: Example (1)

Thread 1
- x = 10;
- wait();
- y = 20;
- (end)

Thread 2
- x = 10;
- (end)

Thread 3
- y = 20;
- (end)

Current state: x = 0, y = 0
Concurrency in SystemC: Example (1)

Current state: $x=10$, $y=0$
Concurrent in SystemC: Example (1)

Current state: \( x = 10, \ y = 0 \)
Concurrency in SystemC: Example (1)

Thread 1

x = 10;
wait();
y = 20;
(end)

Thread 2

x++; (end)

Thread 3

y++; (end)

Current state: x = 10, y = 20
Concurrency in SystemC: Example (1)

Thread 1
- x = 10;
- wait ();
- y = 20;
- (end)

Thread 2
- x++; 
- (end)

Thread 3
- y++; 
- (end)

Current state: x = 11, y = 20
Concurrency in SystemC: Example (1)

Thread 1
x=10;
wait();
y=20;
(end)

Thread 2
x++;
(end)

Thread 3
y++;
(end)

Current state: x=11, y=21
Concurrency in SystemC: Example (2)

Alternative Schedule

Thread 1
- \texttt{x=10;}
- \texttt{wait();}
- \texttt{y=20;}
- (end)

Thread 2
- (end)
- \texttt{x++;}

Thread 3
- (end)
- \texttt{y++;}

Current state: \texttt{x=0, y=0}
Concurrent in SystemC: Example (2)

Alternative Schedule

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=10;</td>
<td>x++;</td>
<td>y++;</td>
</tr>
<tr>
<td>wait();</td>
<td>(end)</td>
<td>(end)</td>
</tr>
<tr>
<td>y=20;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(end)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Current state: x=10, y=0
Concurrency in SystemC: Example (2)

Alternative Schedule

Thread 1
- x=10;
- wait();
- y=20;
- (end)

Thread 2
- x++;  
- (end)

Thread 3
- y++;  
- (end)

Current state: x=11, y=0
Concurrency in SystemC: Example (2)

Alternative Schedule

Thread 1

\[
x = 10; \\
\text{wait ();} \\
y = 20; \\
\text{(end)}
\]

Thread 2

\[
x++; \\
\text{(end)}
\]

Thread 3

\[
y++; \\
\text{(end)}
\]

Current state: \( x = 11, y = 1 \)
Concurrent in SystemC: Example (2)

Alternative Schedule

**Thread 1**

- \(x=10;\)
- \(\text{wait}();\)
- \(y=20;\)
- (end)

**Thread 2**

- \(x++;\)
- (end)

**Thread 3**

- \(y++;\)
- (end)

Current state: \(x=11, y=20\)
Concurrency in SystemC: Example (2)

**Alternative Schedule**

Thread 1
- \( x = 10; \)
- \( \text{wait}(); \)
- \( y = 20; \)
- (end)

Thread 2
- \( x++; \)
- (end)

Thread 3
- \( y++; \)
- (end)

Current state: \( x = 11, y = 20 \)
Concurrency in SystemC

- The example program has a race, i.e., the result depends on the schedule
Concurrency in SystemC

- The example program has a race, i.e., the result depends on the schedule.

- Standard: thread schedule non-deterministic, but must be consistent between simulation runs. → many programmers don’t care about those races.
Concurrency in SystemC

- The example program has a race, i.e., the result depends on the schedule.

- Standard: thread schedule non-deterministic, but must be consistent between simulation runs → many programmers don’t care about those races.

- But: source of error, and simulation/synthesis differences.
Concurrency in SystemC

- The example program has a race, i.e., the result depends on the schedule

- Standard: thread schedule non-deterministic, but must be consistent between simulation runs → many programmers don’t care about those races

- But: source of error, and simulation/synthesis differences

- SystemC offers synchronization constructs to make the schedule deterministic
  - Explicit events
  - FIFOs
  - ...

D. Kroening: FV for better DV
Concurrency in High-Level Modeling

How about concurrency in high-level models?
Concurrency in High-Level Modeling

How about concurrency in high-level models?

FIFO

The ordering of events is crucial. We want the schedule to be non-deterministic! Similar: bus systems, arbiters, ...

D. Kroening: FV for better DV
Concurrent in High-Level Modeling

How about concurrency in high-level models?

CLK 1

FIFO

CLK 2

D. Kroening: FV for better DV
Concurrent in High-Level Modeling

How about concurrency in high-level models?

- The ordering of events is crucial
- We want the schedule to be non-deterministic!
- Similar: bus systems, arbiters, ...

D. Kroening: FV for better DV
Hunting Schedule-Related Bugs

Goal: explore multiple schedules to find schedule-related bugs

- Often done by means of “random” waits
- Not promising due to exponential number of schedules
Goal: explore multiple schedules to find schedule-related bugs

- Often done by means of “random” waits
- Not promising due to exponential number of schedules

- Alternative: try to explore relevant schedules exhaustively
Hunting Schedule-Related Bugs

“ Relevant” schedules?

Observation:

<table>
<thead>
<tr>
<th></th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\text{(end)}

x++;
\text{(end)}

y++;

\text{(end)}

The relative ordering of thread 2 and thread 3 is irrelevant for the state that is finally reached!
Hunting Schedule-Related Bugs

“Relevant” schedules?

Observation:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=10;</td>
<td>x++;</td>
<td>y++;</td>
</tr>
<tr>
<td>wait ();</td>
<td>(end)</td>
<td>(end)</td>
</tr>
<tr>
<td>y=20;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(end)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The relative ordering of thread 2 and thread 3 is irrelevant for the state that is finally reached!
Commutativity of Transitions

Key observation:
\( x++ \) and \( y++ \) are *commutative*
Commutativity of Transitions

Key observation:
\( x++ \) and \( y++ \) are *commutative*
Commutativity of Transitions

Key observation:
\( x++ \) and \( y++ \) are commutative
Commutativity of Transitions

Key observation:
$x++$ and $y++$ are *commutative*

![Diagram showing commutativity of transitions]
Commutativity of Transitions

Key observation:
$x++$ and $y++$ are *commutative*
Commutativity of Transitions

Key observation:
$x++$ and $y++$ are commutative

Idea: explore only one of the paths
This often results in an exponential reduction!
Partial-Order Reduction: Notation

Partial-Order Reduction is a formalization of this idea.

We first define what it means for two threads to be *independent*.

**Definition (Transition System)**

Triple \((S, S_0, \rightarrow)\) where

- \(S\): Set of states,
- \(S_0 \subset S\): set of initial states,
- \(\rightarrow\): set of transitions
- A transition \(\alpha \in \rightarrow\) is a relation on \(S\).

We write \(s \xrightarrow{\alpha} t\) if \((s, t) \in \alpha\).
Partial-Order Reduction: Notation

Thread 1

- \texttt{x=10;}
- \texttt{wait();}
- \texttt{y=20;}
- \texttt{(end)}

Thread 2

- \texttt{x++;}
- \texttt{(end)}

Thread 3

- \texttt{y++;}
- \texttt{(end)}

- Set of states: values of \(x\) and \(y\)
- Set of initial states: \(\{(0, 0)\}\)
- Transitions?
Partial-Order Reduction: Notation

Thread 1
- $x=10$; $\alpha_1$
- wait ()
- $y=20$
- (end)

Thread 2
- $x++;$
- (end)

Thread 3
- $y++;$
- (end)

- Set of states: values of $x$ and $y$
- Set of initial states: $\{(0, 0)\}$
- Transitions?
Partial-Order Reduction: Notation

Set of states: values of x and y
Set of initial states: \{(0, 0)\}
Transitions?

- Thread 1
  - x = 10; $\alpha_1$
  - wait ();
  - y = 20; $\alpha_2$
  - (end)

- Thread 2
  - x++; (end)

- Thread 3
  - y++; (end)
Partial-Order Reduction: Notation

Set of states: values of $x$ and $y$
Set of initial states: $\{(0, 0)\}$
Transitions?

Thread 1
- $x=10$; $\alpha_1$
- wait();
- $y=20$; $\alpha_2$
- (end)

Thread 2
- $x++; \alpha_3$
- (end)

Thread 3
- $y++;$
- (end)
Partial-Order Reduction: Notation

Thread 1

\[ \begin{align*}
x &= 10; & \alpha_1 \\
\text{wait ();} \\
y &= 20; & \alpha_2 \\
(\text{end}) &
\end{align*} \]

Thread 2

\[ \begin{align*}
x &\Rightarrow \alpha_3 \\
(\text{end}) &
\end{align*} \]

Thread 3

\[ \begin{align*}
y &\Rightarrow \alpha_4 \\
(\text{end}) &
\end{align*} \]

- Set of states: values of \( x \) and \( y \)
- Set of initial states: \( \{(0, 0)\} \)
- Transitions?
Partial-Order Reduction: Independence

Extra complication:
   Transitions may be *enabled* (or not).
   E.g., a transition may be sensitive to a clock edge.

We write $\alpha \in \text{Enabled}(s)$ if $\exists s, t. \ s \xrightarrow{\alpha} t$. 
Partial-Order Reduction: Independence

Extra complication:
Transitions may be enabled (or not).
E.g., a transition may be sensitive to a clock edge.

We write \( \alpha \in \text{Enabled}(s) \) if \( \exists s, t. s \xrightarrow{\alpha} t \).

**Definition (Independence)**

Transitions \( \alpha, \beta \) are independent in \( s \) iff

1. \( \alpha \in \text{Enabled}(s) \Rightarrow \beta \in \text{Enabled}(s) \iff \beta \in \text{Enabled}(\alpha(s)) \), and

2. \( \alpha, \beta \in \text{Enabled}(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s)) \)
Partial-Order Reduction

1. **Sleep sets**: maintain set of transitions that can be skipped during the exploration

2. **Persistent sets**: maintain set of transitions which are the only ones to be explored

The techniques are orthogonal, and can be combined.
**Persistent Sets**

Set get_pers(Set runnable)

Set persistents = ∅;

for all (Process \( p_i, p_j \in \text{runnable} \)) do

if (commutative\((p_i, p_j)\)) then

if \((p_i \notin \text{persistents})\) then

persistents := persistents \(\cup\) \{\(p_j\}\);

else

persistents := persistents \(\cup\) \{\(p_i, p_j\}\);

return persistents;
SC_MODULE(m) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == PMAX)
            pressure = PMAX - 1;
    }

    void increment() { pressure++; }

    SC_CTOR(m) {
        SC_METHOD(p2); sensitive << clk;
        SC_METHOD(p1); sensitive << clk;
    }
};
SystemC Running Example

```c
SC_MODULE(m) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == PMAX)
            pressure = PMAX - 1;
    }

    void increment() { pressure++; }

    SCCTOR(m) {
        SC_METHOD(p2); sensitive << clk;
        SC_METHOD(p1); sensitive << clk;
    }
};
```

variable definitions

D. Kroening: FV for better DV
SystemC Running Example

```
SC_MODULE(m) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == PMAX)
            pressure = PMAX - 1;
    }

    void increment() {
        pressure++;
    }

    SC_CTOR(m) {
        SC_METHOD(p2); sensitive << clk;
        SC_METHOD(p1); sensitive << clk;
    }
};
```
SystemC Running Example

```c
SC_MODULE(m) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == PMAX)
            pressure = PMAX - 1;
    }

    void increment() { pressure++;

    SCCTOR(m) {
        SC_METHOD(p2); sensitive << clk;
        SC_METHOD(p1); sensitive << clk;
    }
};
```

- variable definitions
- two thread definitions
- constructor – sets up threads
Intent: Keep PMAX lower than pressure

After each clock tick, the scheduler can choose between

(guard, increment) and (increment, guard)

There is an exponential number of possibilities (in the number of clock ticks)
Running Example

- Intent: Keep PMAX lower than pressure
- After each clock tick, the scheduler can choose between
  \[(\text{guard, increment}) \quad \text{and} \quad (\text{increment, guard})\]
- There is an exponential number of possibilities (in the number of clock ticks)

\[\text{the property can be violated with the right schedule}\]
Detecting Commutativity

- Commutativity is typically approximated by means of static analysis

- E.g., compute set of read/written variables

- Works well for fine-grained parallelism
Related Work

Alper Sen, Vinit Ogale, Magdy S. Abadir: 
*Predictive Runtime Verification of Multi-Processor SoCs in SystemC* 
(DAC 2008)

Idea:

- Modify the SystemC simulation kernel
- Keep track of usage of synchronization primitives
- Analyze the resulting partial-order trace for potential violations of the property

But: what if the synchronization performed depends on previous scheduling?
Related Work

Sudipta Kundu, Malay Ganai, Rajesh Gupta:
*Partial Order Reduction for Scalable Testing of SystemC TLM Designs*
(DAC 2008)

Idea:

- Perform static analysis to detect writes on shared variables/signals, and waiting with immediate notification

- Use this to compute a dependence relation

- Modify scheduler accordingly
SystemC Running Example

Are these independent?

```c
void guard() {
    if (pressure == PMAX)
        pressure = PMAX - 1;
}

void increment() {
    pressure++;
}
```
Are these independent?

```c
void guard() {
    if (pressure == PMAX)
        pressure = PMAX - 1;
}
```

```c
void increment() {
    pressure++;
}
```

No, because they are not commutative.
SystemC Running Example

Are these independent?

```c
void guard() {
    if (pressure == PMAX)
        pressure = PMAX - 1;
}
```

```c
void increment() {
    pressure++;
}
```

No, because they are not commutative.

Partial order reduction with usual static commutativity analysis yields no reduction.
Partial-Order Reduction for Big-Step Parallelism

- High-level models typically have large blocks that are executed atomically.
- There are very few independent transitions.
- Remedy: compute dependency relation separately for each state.
SystemC Running Example

Are these independent?

```c
void guard() {
    if (pressure == PMAX)
        pressure = PMAX - 1;
}

void increment() {
    pressure++;  // This line is incorrect, pressure++ is not valid.
}
```

Yes, in *most* states!
Scoot: Key Idea

- **Scoot** is an analyzer for SystemC models

- Key idea: use formal verification (FV) to compute the set of states in which a pair of transactions is commutative

- Then re-synthesize the code to obtain a statically scheduled simulator for dynamic verification (DV)

→ Removes enormous overhead
An Overview of Scoot

Simplified version of the SystemC header files: systemc.h

User-provided SystemC models

Typechecker
Control-Flow Graph
Pointer Analysis
Module-Hierarchy Analysis
Race-Condition Analysis
Scheduler Synthesis
Code Re-synthesis

Scoot

Flat C++ Model

\texttt{g++}

Exhaustive Simulator

D. Kroening: FV for better DV
Scheduling with Commutativity Information

- We only consider *time notification* and *delta notification*

- Thus, we ban immediate notification
  \[ \rightarrow \text{processes cannot enable each other during the evaluation phase} \]
Scheduling with Commutativity Information

- We only consider *time notification* and *delta notification*

- Thus, we ban immediate notification
  \[ \rightarrow \text{ processes cannot enable each other during the evaluation phase} \]

- Consequence:

  \[ \text{Runnable}(\alpha(s)) = \text{Runnable}(s) \setminus \{\alpha\} \]
Scheduling with Commutativity Information

Theorem (valid without immediate notification):

Two processes $\alpha, \beta \in \text{Runnable}(s)$ are independent in $s$ if they are commutative in $s$.

It’s enough to find out if two processes are commutative.
Model Checking

- **Model Checking** is an algorithmic technique for verifying if a given model satisfies a given property.

- Relies on an exhaustive analysis of the state space of the model.

- Frequently applied in the hardware domain.

- Key value: diagnostic counterexamples in case the property does not hold.

- But: susceptible to state-space explosion problem.

D. Kroening: FV for better DV
Model Checking with Predicate Abstraction

- A heavy-weight formal analysis technique

- Recent successes in software verification, e.g., SLAM at Microsoft

- The abstraction reduces the size of the model by removing irrelevant detail

- Idea: only track predicates on data, and remove data variables from model

- Mostly applies to control-flow dominated properties
Model Checking with Predicate Abstraction

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i++;
}
```

C Program
Model Checking with Predicate Abstraction

```c
int main() {
    int i;
    i = 0;
    while (even(i))
        i++;
}
```

C Program

Predicate

$p_1 \iff i = 0$

$p_2 \iff \text{even}(i)$

D. Kroening: FV for better DV
Model Checking with Predicate Abstraction

```c
int main() {
    int i;
i = 0;
    while (even(i))
i++;
}
```

4

```
void main() {
    bool p1, p2;
p1 = TRUE;
p2 = TRUE;
    while (p2) {
        p1 = p1 ? FALSE : p1;
p2 = !p2;
    }
}
```

D. Kroening: FV for better DV
Counterexample-guided Abstraction Refinement

1. Abstract
2. Check abstraction
3. Property holds on abstraction: terminate
4. Otherwise: refine predicates and start over
Computing the Commutativity Condition

Idea: build assertion that captures commutativity
Computing the Commutativity Condition

Idea: build assertion that captures commutativity

```
assume(\phi);
2 s_0 := current_state;
p_1 (); p_2 ();
4 s_{1,2} := current_state;
current_state := s_0;
6 p_2 (); p_1 ();
s_{2,1} := current_state;
8 assert(s_{1,2} \neq s_{2,1});
```

Counterexamples show paths starting in states in which the processes are commutative
Computing the Commutativity Condition

1. Start with $\phi = true$
2. Apply model checker to harness
3. If assertion unreachable, terminate
4. The Model Checker produces a counterexample trace $\pi$
5. Otherwise, record *weakest precondition* $P_\pi$ of $s_{1,2} = s_{2,1}$ along $\pi$
6. Strengthen $\phi$ to remove states satisfying $P_\pi$

The predicate $\bigvee_\pi P_\pi$ is the weakest condition such that $p_1/p_2$ are commutative
Running Example Again

```
assume(\phi);
2 \quad s_0 := \text{pressure};

4 \quad \textbf{if} (\text{pressure}==\text{PMAX}) \quad \text{pressure}=\text{PMAX}−1; \quad \text{// } p_1();
\quad \text{pressure}++; \quad \text{// } p_2();

6 \quad s_{1,2} := \text{pressure};
8 \quad \text{pressure} := s_0;

10 \quad \text{pressure}++; \quad \text{// } p_2();
\quad \textbf{if} (\text{pressure}==\text{PMAX}) \quad \text{pressure}=\text{PMAX}−1; \quad \text{// } p_1();

12 \quad s_{2,1} := \text{pressure};
14 \quad \text{assert}(s_{1,2} \neq s_{2,1});
```
Running Example Again

```plaintext
assume(\phi);
2 \ s_0 := pressure;

4 \textbf{if} (\text{pressure}==\text{PMAX}) \text{ pressure}=\text{PMAX}-1; // p_1();
\text{ pressure}++; // p_2();

6 \ s_{1,2} := pressure;
8 \text{ pressure} := \ s_0 ;

10 \text{ pressure}++; // p_2();
\textbf{if} (\text{pressure}==\text{PMAX}) \text{ pressure}=\text{PMAX}-1; // p_1();

12 \ s_{2,1} := pressure;
14 \text{ assert}(s_{1,2} \neq s_{2,1});

\textbf{pressure} \neq \text{PMAX} - 1 \land \text{ pressure} \neq \text{PMAX}
```
Integration into CEGAR

Concrete program

Abstract prog.

Model Checking

Abstract trace

No trace

End

Concrete program

Abstraction

Simulation

Spurious trace

Concrete trace

Refinement

Strengthening $\phi$

New predicates

$\phi'$
Experimental Results

Benchmarks:
- B1/B2
- Both: three processes (one server, two clients)
- Typical instances for transaction-level modeling
Results on Benchmark B1

![Graphs showing # Transitions vs # Simulation steps and Time (s) vs # Simulation steps for different scenarios: No-POR, P, S, P+S.]
Results on Benchmark B2

- **# Transitions**
  - No-POR
  - P
  - S
  - P+S

- **Time (s)**
  - No-POR
  - P
  - S
  - P+S

D. Kroening: **FV for better DV**
The Price to Pay

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Pair</th>
<th>SATABS [s]</th>
<th># Strengthenings</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>0</td>
<td>&lt;1</td>
<td>3</td>
</tr>
<tr>
<td>B1</td>
<td>1</td>
<td>23</td>
<td>17</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>21</td>
<td>17</td>
</tr>
<tr>
<td>B2</td>
<td>0</td>
<td>1111</td>
<td>65</td>
</tr>
<tr>
<td>B2</td>
<td>1</td>
<td>396</td>
<td>24</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
<td>638</td>
<td>23</td>
</tr>
</tbody>
</table>

Model Checker only sees pairs of transitions

Pairs can be distributed on a cluster!
Conclusion

- SystemC hottest candidate as new architectural high-level specification language

- New opportunities for formal analysis!

- If you are running lots of simulations, it may be worthwhile to perform some heavy static analysis first