Race Analysis for SystemC using Model Checking

Nicolas Blanc, Daniel Kroening
Outline

Motivation

Partial-Order Reduction

Scoot

Experimental Results
Introduction

- Oxford is a major verification center: 9 full-time academics, 30+ post-docs

- My group
  - 10 PhD students, 8 post-docs (hiring 2 more)
  - GBP 3m funding for verification (industry, Artemis, FP7)
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

D. Kroening: Race Analysis for SystemC Using Model Checking
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

- 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**

**g++** → **Executable**

+ **Stimulus**
  = **Traces**

D. Kroening: *Race Analysis for SystemC Using Model Checking*
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
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
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
    }
}
```

D. Kroening: Race Analysis for SystemC using Model Checking
SystemC

```c
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
```

C/C++

Verilog/VHDL

Convenient modeling of both hardware and software

D. Kroening: Race Analysis for SystemCusing Model Checking
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
  - But: Interleaving only at specific locations
  - \texttt{wait()}
  - End of thread

→ No issues with atomicity
→ Doesn’t really map onto usual pthread model

Makes synthesis and model checking much easier!
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!
Concurrent SystemC: Example (1)

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

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

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

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

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

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

Current state: $x=11$, $y=21$
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: \( x=0, y=0 \)
Concurrent in SystemC: Example (2)

Alternative Schedule

Thread 1

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

Thread 2

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

Thread 3

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

Current state: \( x=10, y=0 \)
Concurrent 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;
- wait ();
- y=20;
- (end)

Thread 2
- x++;  
- (end)

Thread 3
- y++;  
- (end)

Current state: x=11, y=1

D. Kroening: Race Analysis for SystemC using Model Checking
Concurrent 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=1
Concurrency in SystemC: Example (2)

Alternative Schedule

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

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
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: Race Analysis for SystemC using Model Checking
Concurrency in High-Level Modeling

How about concurrency in high-level models?
Concurrency 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: *Race Analysis for SystemCusing Model Checking*
Concurrent in High-Level Modeling

How about concurrency in high-level models?

D. Kroening: Race Analysis for SystemCusing Model Checking
Concurrency 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: Race Analysis for SystemC using Model Checking
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
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

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

“Relevant” schedules?
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

$S_1$
Commutativity of Transitions

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

D. Kroening: *Race Analysis for SystemCusing Model Checking*
Commutativity of Transitions

Key observation:

$x++$ and $y++$ are *commutative*
Commutativity of Transitions

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

A
A
A
A U
A
A
A U
A
A
A U
A
A
A U
A
A
A U

Idea: explore only one of the paths
This often results in an exponential reduction!
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

<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>

- 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

$\alpha_1$

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

Thread 2

x++;  
(end)

Thread 3

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

Thread 1

\[\begin{array}{l}
\text{x=10;} \\
\text{wait();} \\
\text{y=20;} \\
\text{(end)} \end{array} \text{ } \alpha_1 \]

Thread 2

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

Thread 3

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

- 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; $\alpha_2$
- (end)

Thread 2

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

Thread 3

- y++; (end)

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x=10; \quad \alpha_1)</td>
<td>(x++; \quad \alpha_3)</td>
<td>(y++; \quad \alpha_4)</td>
</tr>
<tr>
<td><code>wait ();</code></td>
<td>(end)</td>
<td>(end)</td>
</tr>
<tr>
<td>(y=20; \quad \alpha_2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(end)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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 $\exists t. s \xrightarrow{\alpha} t$. 
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 $\exists t. s \xrightarrow{\alpha} t$.

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

1. `Set get_pers(Set runnable)`

2. `Set persistents = ∅;`

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

4. `if (commutative(\( p_i, p_j \))) then`

5. `if (\( p_i \notin \text{persistents} \)) then`

6. `persistents := persistents \cup \{p_j\};`

7. `else`

8. `persistents := persistents \cup \{p_i, p_j\};`

9. `return persistents;`
SystemC Running Example

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

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

    void increment() { pressure++; }

    SC_CTOR(m) {
        SC_METHOD(guard); sensitive << clk;
        SC_METHOD(increment); sensitive << clk;
    }
};
```

D. Kroening: Race Analysis for SystemC Using Model Checking
SystemC Running Example

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

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

    void increment() {
        pressure++;
    }

    SC_CTOR(m) {
        SC_METHOD(guard); sensitive << clk;
        SC_METHOD(increment); sensitive << clk;
    }
};
```
SystemC Running Example

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

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

  void increment() { pressure++; }

  SC_CTOR(m) {
    SC_METHOD(guard); sensitive << clk;
    SC_METHOD(increment); 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++;
    }

    SC_CTOR(m) {
        SC_METHOD(guard); sensitive << clk;
        SC_METHOD(increment); sensitive << clk;
    }
};
```

variable definitions

two thread definitions

constructor – sets up threads
Running Example

- Intent: Keep pressure lower than PMAX
- 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 pressure lower than PMAX

- After each clock tick, the scheduler can choose between 
  \[(\text{guard, increment}) \text{ and } (\text{increment, guard})\]

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

⚠️ 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
SystemC Running Example

Are these independent?

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

```c
void increment() {
    pressure++;
}
```
SystemC Running Example

Are these independent?

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

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++;
}
```

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!

D. Kroening: *Race Analysis for SystemC Using Model Checking*
SystemC Running Example

Are these independent?

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

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

![Diagram showing the states and transitions]

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in most states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

Yes, in *most* states!
SystemC Running Example

Are these independent?

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

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

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

g++

Exhaustive Simulator

D. Kroening: Race Analysis for SystemC using Model Checking
We only consider *time notification* and *delta notification*

Thus, we ban immediate notification
→ 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:
  \[
  \text{Runnable}(\alpha(s)) = \text{Runnable}(s) \setminus \{\alpha\}
  \]
Theorem (valid without immediate notification):

Two processes \( \alpha, \beta \in \text{Runnable}(s) \) are independent in \( s \) if they are \underline{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
Experimental Results

Benchmarks:

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

---

[Graphs showing the comparison of # Transitions and Time (s) for different methods: No-POR, P, S, and P+S. The graphs illustrate the simulation steps and time taken as the number of steps increases.]
Results on Benchmark B2
# The Price to Pay

Model Checker only sees **pairs of transitions**

Pairs can be distributed on a cluster!

<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>

D. Kroening: Race Analysis for SystemCusing Model Checking
Conclusion

- SystemC offers *new opportunities for formal analysis!*

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

- Please talk to us if you are interested in
  - co-verification (e.g. driver + Verilog RTL)
  - SystemC