Race Analysis for SystemC using Model Checking
Nicolas Blanc, Daniel Kroening

www.cprover.org/scoot

Supported by Intel and SRC
This Talk

Formal race analysis for SystemC and its applications beyond verification.

- Race analysis: based on Model Checking
- Exhaustive Simulation: Partial-Order Reduction
Race

- Race in a **concurrent system**: the outcome of the computation depends on the scheduling

- Mechanism to model nondeterminism implicitly
  - Design flaw: state corruption, deadlock,…
  - Hard to verify: combinatorial explosion of schedules

- **SystemC**: language based on C++ for modeling concurrent systems
The SystemC Scheduler

- Cooperative Multitasking Semantics:
  - One process running at a time
  - No preemption
- Execution driven by *events*
- Two kind of processes:
  - *method process*: forbidden to suspend its execution
  - *thread*: can wait for event notifications
Introductory Example

SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == 10)
            pressure = 9;
    }

    void increment() {
        pressure++;
    }

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

• Two processes: guard, increment
• Shared variable: pressure

• Number of traces grows exponentially with the simulation time

• Pressure can exceed the limit.
SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == 10)
            pressure = 9;
    }

    void increment() {
        pressure++;
    }

    SC_CTOR(Module) {
        SC_METHOD(guard);
        sensitive << clk;
        SC_METHOD(increment);
        sensitive << clk;
    }
};
SC_MODULE(Module) {
  sc_clock clk;
  int pressure;

  void guard() {
    if (pressure == 10)
      pressure = 9;
  }

  void increment() {
    pressure++;
  }
}

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

Classic static analysis achieves no reduction!
SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == 10)
            pressure = 9;
    }

    void increment() {
        pressure++;
    }

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

Dynamic Partial-Order Reduction:

Write

increment

Read

guard
SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard()
    {
        if (pressure == 10)
            pressure = 9;
    }

    void increment()
    {
        pressure++;
    }

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

Dynamic Partial-Order Reduction:

Alternative Schedule

Runtime analysis achieves no reduction!
When are the processes independent?

SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard() {
        if (pressure == 10)
            pressure = 9;
    }

    void increment() {
        pressure++;
    }

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

\[ \phi \leftrightarrow \text{pressure} \neq 9 \land \text{pressure} \neq 10 \]
Guarded Independence \cite{C. Wang et al., TACAS 2008}

Derives directly from P. Godefroid’s notion of conditional independence.

Definition. Two transitions $\alpha$ and $\beta$ are guarded independent with respect to a guard $\phi \subseteq S$ if and only if for all $s \in \phi$ the following hold:

1. $\alpha \in Enabled(s) \Rightarrow \\
   \beta \in Enabled(s) \Leftrightarrow \beta \in Enabled(\alpha(s))$

2. $\beta \in Enabled(s) \Rightarrow \\
   \alpha \in Enabled(s) \Leftrightarrow \alpha \in Enabled(\beta(s))$

3. $\alpha, \beta \in Enabled(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$

- Contribution:
  - Formal technique to compute $\phi$ for SystemC

- Applications: formal verification, simulation, and testing
Exploration with Partial-Order Reduction

SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    void guard() {
        if(pressure == 10)
            pressure = 9;
    }

    void increment() {
        pressure++;
    }

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

$\phi \iff pressure \neq 9 \land pressure \neq 10$
Computation of Guarded Independence

**IDEA**: exploit the cooperative execution model of SystemC to compute \( \phi \) iteratively using a harness.

\[
\begin{align*}
    s_0 & := \text{current\_state}; \\
    p_1 () ; p_2 () ; \\
    s_{1,2} & := \text{current\_state}; \\
    \text{current\_state} & := s_0 ; \\
    p_2 () ; p_1 () ; \\
    s_{2,1} & := \text{current\_state}; \\
    \text{assert} ( s_{1,2} \neq s_{2,1} );
\end{align*}
\]

We have found a trace such that:
1. the execution of \( p_1, p_2 \) terminates, and
2. the order of execution is irrelevant.

Counterexample
Using the Counterexample

Weakest precondition:

\[ P_\pi = \text{wp}(\pi, s_{1,2} = s_{2,1}) \]

If \( P_\pi \) holds in the pre-state, then:
1. the execution terminates, and
2. the execution order is irrelevant.

\( P_\pi \) is an under-approximation of \( \phi \).

States
Computing Preconditions

Hoare’s rule for assignments:

\[ \{ P[v/E] \} \quad v := E; \quad \{ P \} \]

```plaintext
\{ y < x \}
  tmp := x;
\{ y < tmp \}
x := y;
\{ x < tmp \}
y := tmp;
\{ x < y \}
```

Precondition  Command  Postcondition
Strengthening the set of initial states

Remove $\pi$ using $P_\pi$

```
assume(\neg P_\pi);
s_0 := \text{current\_state};
p_1(); \ p_2();
s_{1,2} := \text{current\_state};
current\_state := s_0;
p_2(); \ p_1();
s_{2,1} := \text{current\_state};
assert(s_{1,2} \neq s_{2,1});
```
Automated Procedure

set to true initially

assume (τ);
s₀ := current_state;
p₁(); p₂();
s₁₂ := current_state;
current_state := s₀;
p₂(); p₁();
s₂₁ := current_state;
assert (s₁₂ ≠ s₂₁);

Strengthening Loop:

Begin

Model Checking

φ ⇔ ¬τ

Correct

End

Counterexample π

Strengthening τ

τ′
Scoot: Research Compiler for SystemC

- 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

- Flat C++ Model

- Exhaustive Simulator

\[ g++ \]
Benchmark: Memory Components

- Often subject to race phenomenon
- Present in most electronics designs:
  - FIFOs, bridges, processors
Asynchronous Dual Port RAM

SC_MODULE (ram_dp_ar_aw) {

sc_uint <DATA_WIDTH> mem [RAM_DEPTH];
void READ_0 ();
void WRITE_0 ();
void READ_1 ();
void WRITE_1 ();
};

void READ_0 () {
    if (cs_0.read() && oe_0.read() && !we_0.read())
        data_0 = mem[address_0.read()];
}

void WRITE_0 () {
    if (cs_0.read() && we_0.read())
        mem[address_0.read()] = data_0.read();
}

...
## Benchmark Results

<table>
<thead>
<tr>
<th>Processes</th>
<th>Indep?</th>
<th># Stren.</th>
<th>Predicates</th>
<th>SMV</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd1, Rd0</td>
<td>yes</td>
<td>16</td>
<td>23</td>
<td>13s</td>
<td>40s</td>
</tr>
<tr>
<td>Wr0, Rd0</td>
<td>yes</td>
<td>5</td>
<td>38</td>
<td>12s</td>
<td>50s</td>
</tr>
<tr>
<td>Wr0, Rd1</td>
<td>no</td>
<td>12</td>
<td>169</td>
<td>17m 6s</td>
<td>28m</td>
</tr>
<tr>
<td>Wr1, Rd0</td>
<td>no</td>
<td>12</td>
<td>169</td>
<td>28m 38s</td>
<td>39m 18s</td>
</tr>
<tr>
<td>Wr1, Rd1</td>
<td>yes</td>
<td>5</td>
<td>38</td>
<td>12s</td>
<td>53s</td>
</tr>
<tr>
<td>Wr1, Wr0</td>
<td>no</td>
<td>9</td>
<td>104</td>
<td>38s</td>
<td>5m 24s</td>
</tr>
<tr>
<td>run, Rd0</td>
<td>yes</td>
<td>12</td>
<td>69</td>
<td>3m</td>
<td>3m 47s</td>
</tr>
<tr>
<td>run, Rd1</td>
<td>yes</td>
<td>12</td>
<td>69</td>
<td>2m 47s</td>
<td>3m 35s</td>
</tr>
<tr>
<td>run, Wr0</td>
<td>yes</td>
<td>9</td>
<td>75</td>
<td>2m 17s</td>
<td>3m 35s</td>
</tr>
<tr>
<td>run, Wr1</td>
<td>yes</td>
<td>9</td>
<td>75</td>
<td>2m 17s</td>
<td>3m 35s</td>
</tr>
</tbody>
</table>

\[ \phi \iff \text{true} \]

Linux, Intel Xeon 3GHz.
## Trading Precision for Time

<table>
<thead>
<tr>
<th>Processes</th>
<th>Indep?</th>
<th># Stren.</th>
<th>Predicates</th>
<th>SMV</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wr0, Rd1</td>
<td>no</td>
<td>12</td>
<td>169</td>
<td>17m 6s</td>
<td>28m</td>
</tr>
<tr>
<td>Wr0, Rd1</td>
<td>no</td>
<td>11</td>
<td>42</td>
<td>27s</td>
<td>1m 30s</td>
</tr>
<tr>
<td>Wr1, Rd0</td>
<td>no</td>
<td>12</td>
<td>169</td>
<td>28m 38</td>
<td>39m 18s</td>
</tr>
<tr>
<td>Wr1, Rd0</td>
<td>no</td>
<td>11</td>
<td>42</td>
<td>27s</td>
<td>1m 30s</td>
</tr>
</tbody>
</table>
Due to the combinatorial explosion of schedules, the time spent for static analysis is rapidly amortized!
Conclusion

- “Formal” has applications beyond property checking: optimization, simulation, testing
- **Partition** the system into “small” verification tasks
- **Distribute** those tasks among many cores
- **Trade** precision for time
- **Pragmatic approach** to successful application of formal engines at high abstraction levels
Related Work

- “Partial order reduction for scalable testing of SystemC TLM designs”, Sudipta Kundu et al., DAC 2008
- “Automatic generation of schedulings for improving the test coverage of systems-on-a-chip”, Claude Helmstetter et al., FMCAD 2006
- “Dynamic partial-order reduction for model checking software”, Cormac Flanagan et al., SIGPLAN, 2005
- Patrice Godefroid’s PhD thesis, 1994
Thank you for your attention.