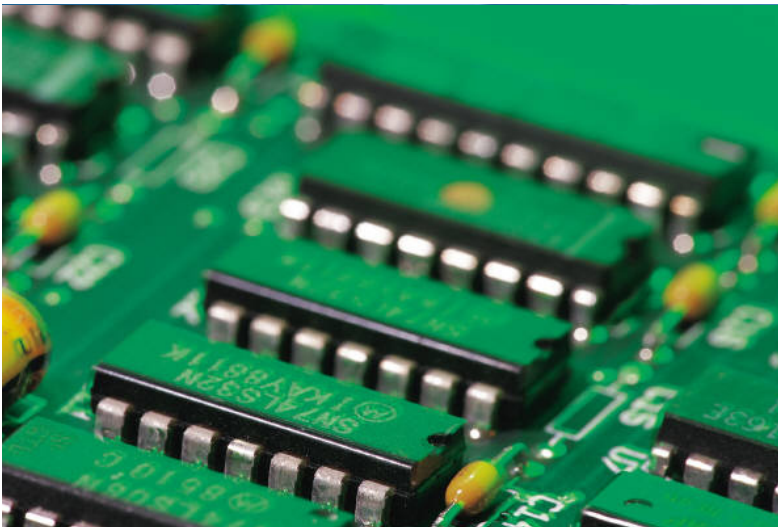


# Race Analysis for SystemC using Model Checking

Nicolas Blanc, Daniel Kroening

[www.cprover.org/scoot](http://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++;  
    }  
};
```

- Two processes: guard, increment
- Shared variable: **pressure**
- Number of traces grows exponentially with the simulation time
- Pressure can exceed the limit.

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



sensitive to the clock

```

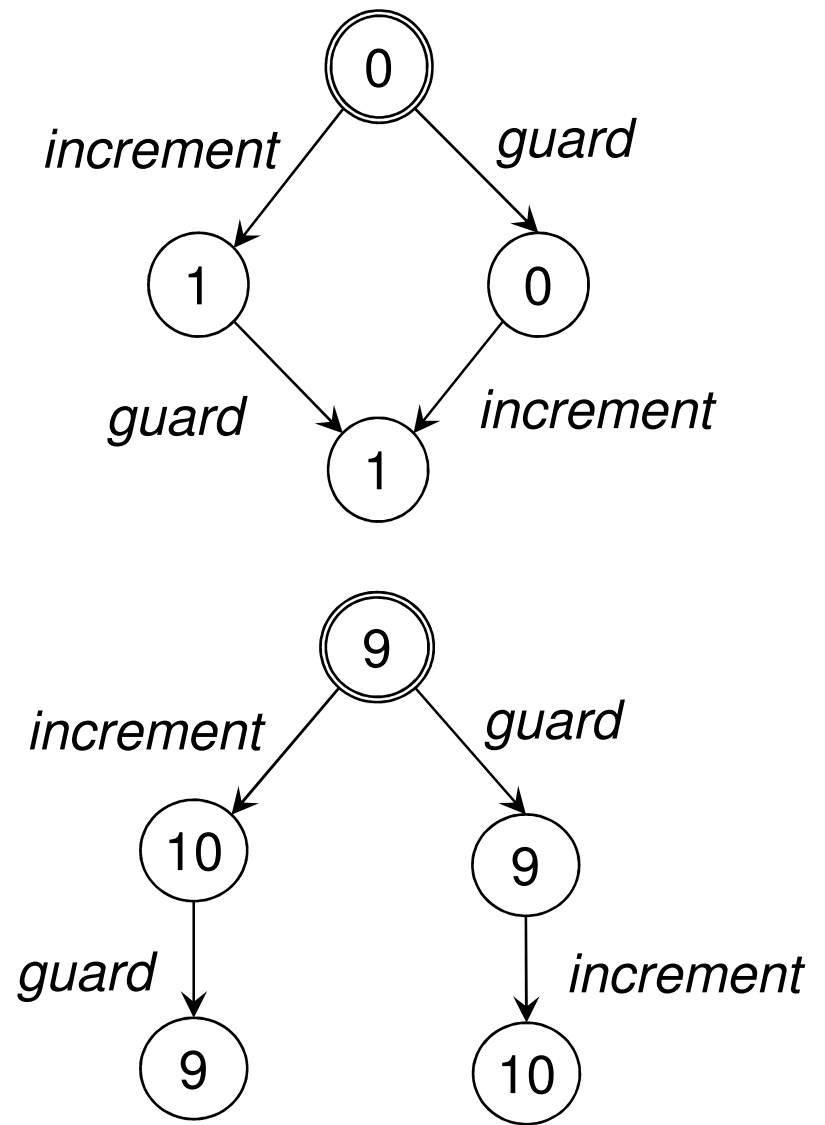
SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

    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;
    }
};
```

Read

Write

Classic static analysis achieves  
no reduction!

```

SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

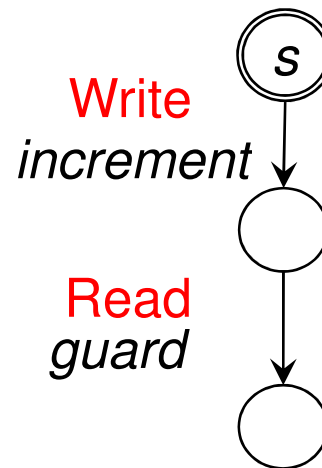
    void guard(){
        if(pressure == 10)
            pressure = 9;
    }

    void increment(){
        pressure++;
    }

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

```

## Dynamic Partial-Order Reduction:





```

SC_MODULE(Module) {
    sc_clock clk;
    int pressure;

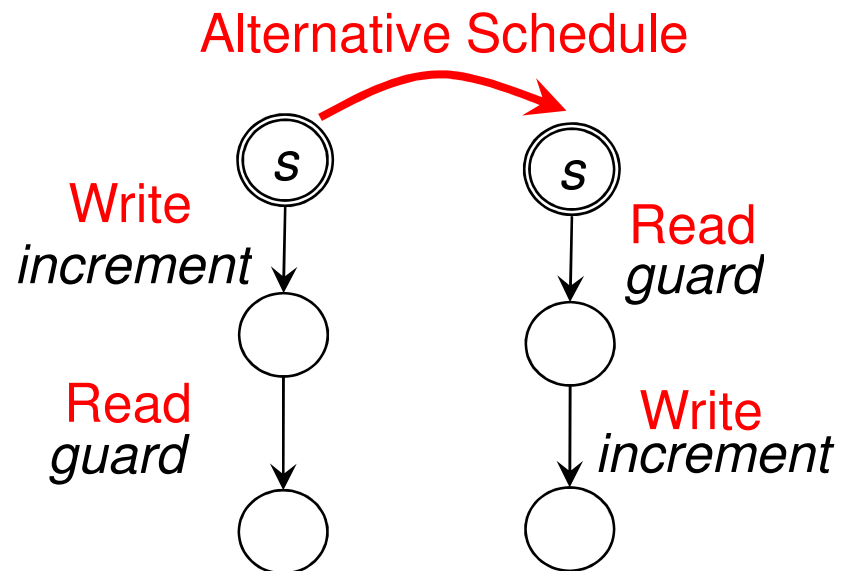
    void guard(){
        if(pressure == 10)
            pressure = 9;
    }

    void increment(){
        pressure++;
    }

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

```

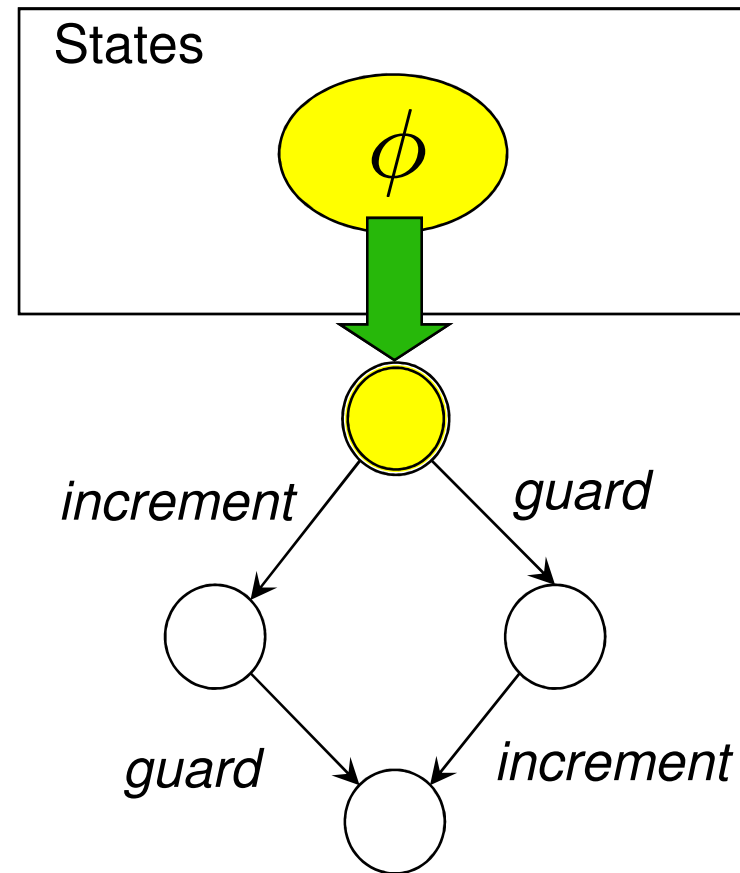
## Dynamic Partial-Order Reduction:



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;  
    }  
};
```



$$\emptyset \Leftrightarrow \text{pressure} \neq 9 \wedge \text{pressure} \neq 10$$

# Guarded Independence [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:

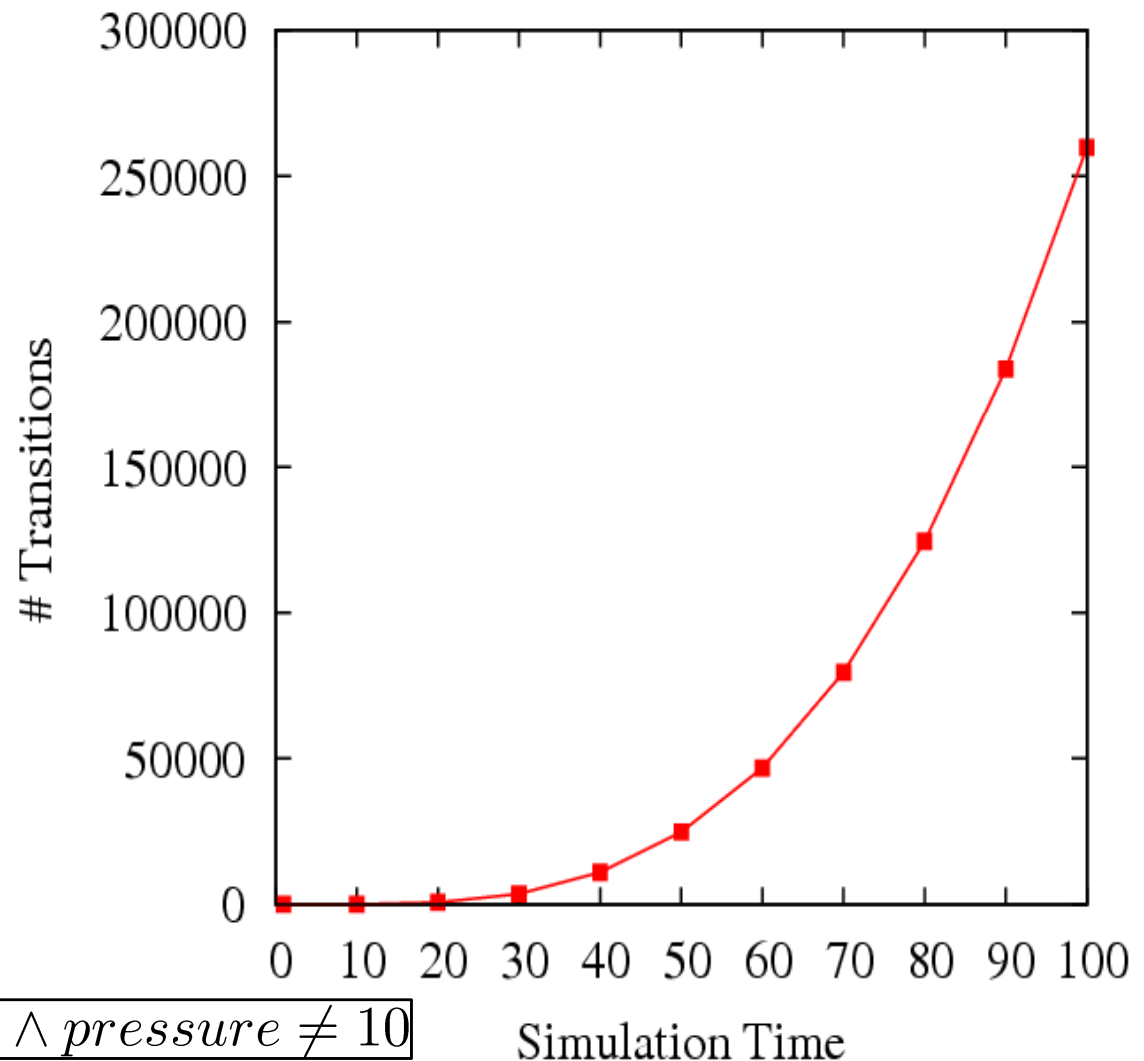
1.  $\alpha \in \text{Enabled}(s) \Rightarrow$   
 $\beta \in \text{Enabled}(s) \Leftrightarrow \beta \in \text{Enabled}(\alpha(s))$
2.  $\beta \in \text{Enabled}(s) \Rightarrow$   
 $\alpha \in \text{Enabled}(s) \Leftrightarrow \alpha \in \text{Enabled}(\beta(s))$
3.  $\alpha, \beta \in \text{Enabled}(s) \Rightarrow \alpha(\beta(s)) = \beta(\alpha(s))$

- 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;  
    }  
};
```

$\phi \Leftrightarrow \text{pressure} \neq 9 \wedge \text{pressure} \neq 10$



# Computation of Guarded Independence

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

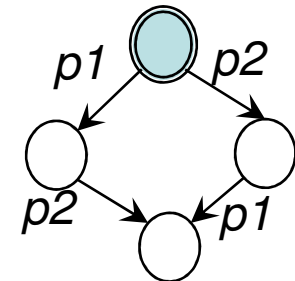
```
s0 := current_state;  
p1(); p2();  
s1,2 := current_state;  
current_state := s0;  
p2(); p1();  
s2,1 := current_state;  
assert(s1,2 ≠ s2,1);
```



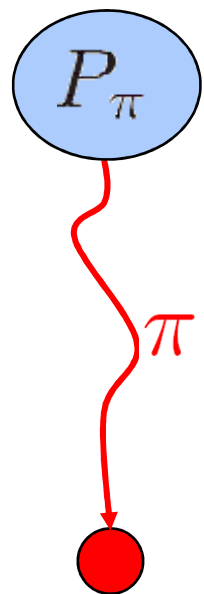
Counterexample

We have found a trace such that:

1. the execution of *p1*, *p2* terminates, and
2. the order of execution is irrelevant.



# Using the Counterexample

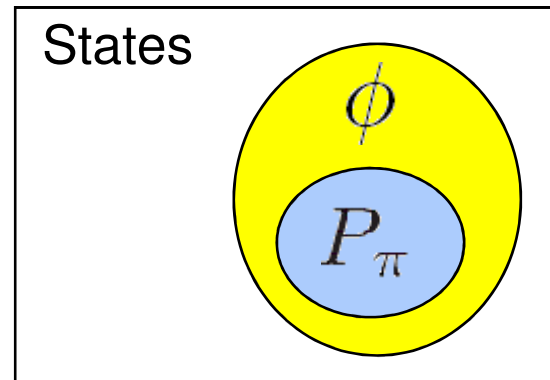


Weakest precondition:

$$P_\pi = wp(\pi, s_{1,2} = s_{2,1})$$

- 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  $\phi$  :



# Computing Preconditions

Hoare's rule for assignments:

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

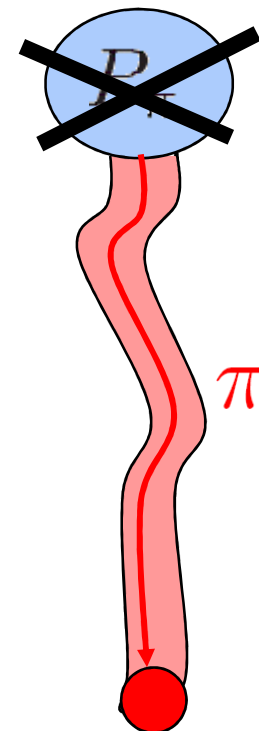
Precondition  Command  Postcondition 

$$\begin{array}{l} \{y < x\} \\ \text{tmp} := x; \\ \{y < \text{tmp}\} \\ x := y; \\ \{x < \text{tmp}\} \\ y := \text{tmp}; \\ \{x < y\} \end{array}$$

# Strengthening the set of initial states

Remove  $\pi$  using  $P_\pi$

```
assume( $\neg P_\pi$ );  
s0 := current_state;  
p1(); p2();  
s1,2 := current_state;  
current_state := s0;  
p2(); p1();  
s2,1 := current_state;  
assert(s1,2 ≠ s2,1);
```



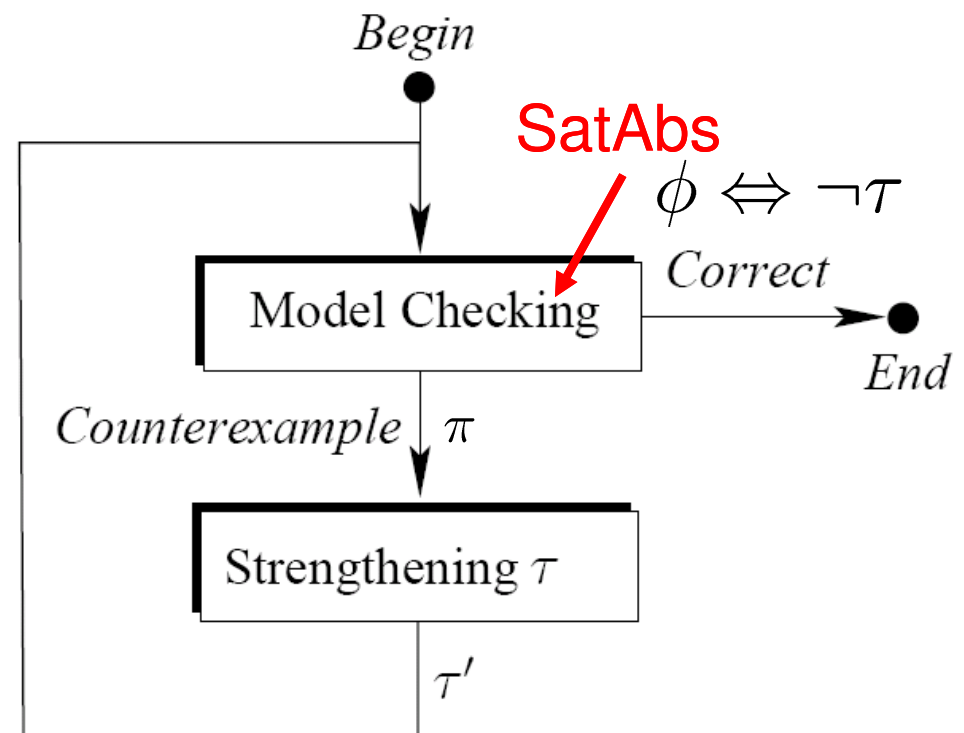


# Automated Procedure

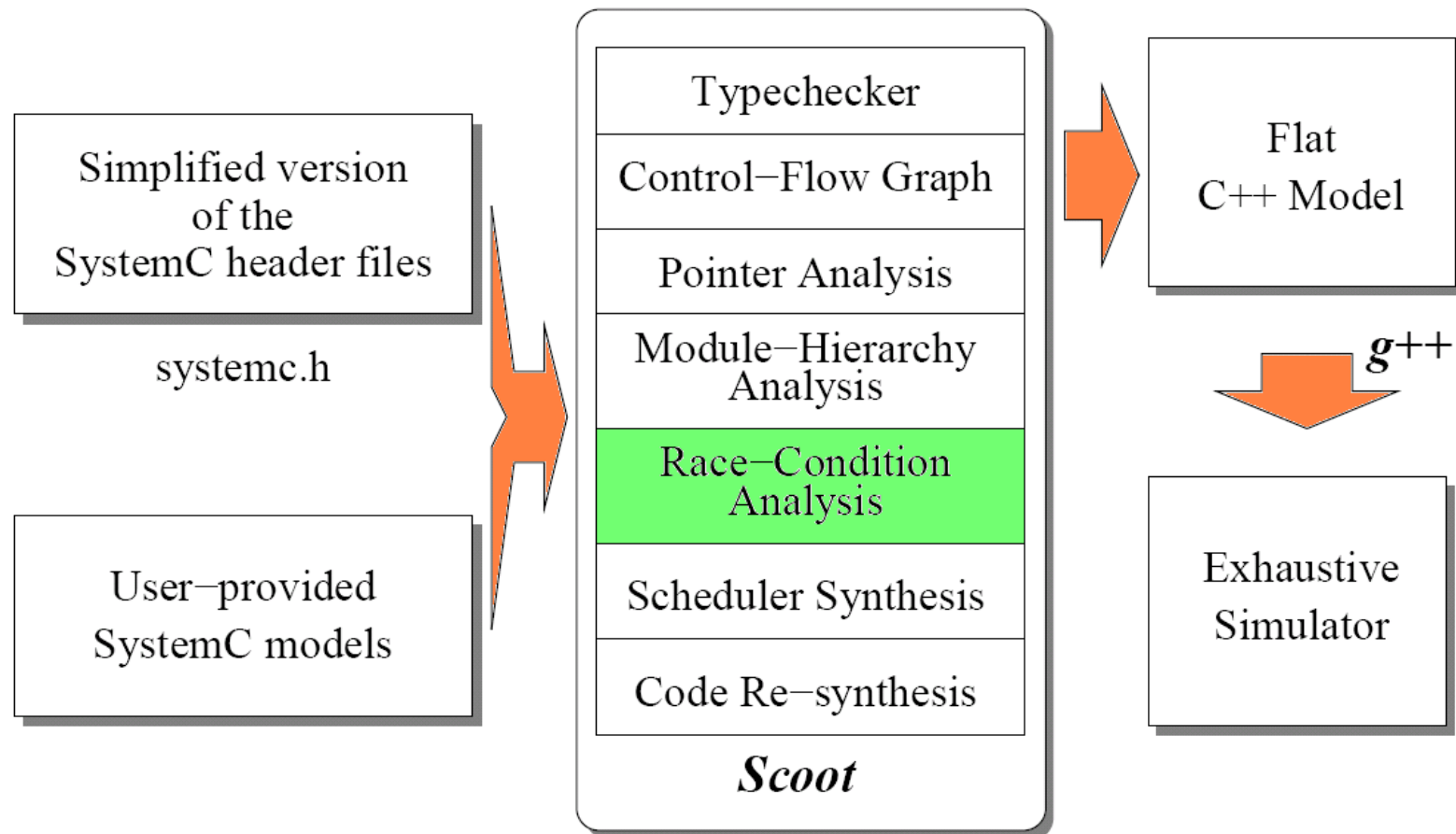
set to true initially

```
assume( $\tau$ );  
 $s_0 := \text{current\_state}$ ;  
 $p_1()$ ;  $p_2()$ ;  
 $s_{1,2} := \text{current\_state}$ ;  
 $\text{current\_state} := s_0$ ;  
 $p_2()$ ;  $p_1()$ ;  
 $s_{2,1} := \text{current\_state}$ ;  
assert( $s_{1,2} \neq s_{2,1}$ );
```

Strengthening Loop:

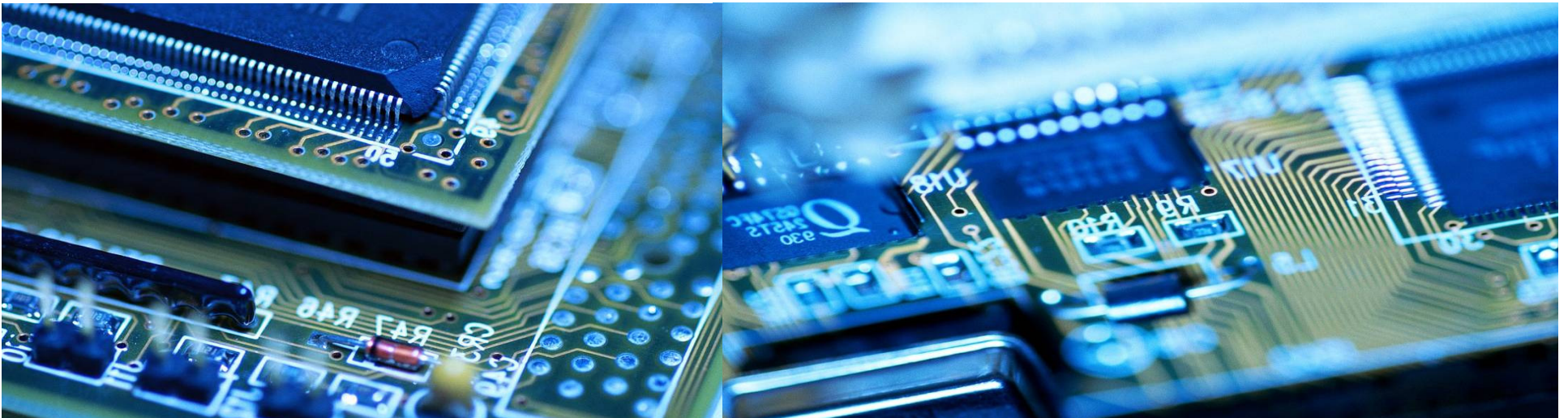


# Scoot: Research Compiler for *SystemC*



# 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();  
}  
...  
}
```

← shared memory

← 2x2 processes

← exclusive RD/WR

# Benchmark Results

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

$\phi \Leftrightarrow true$  ↑

Linux, Intel Xeon 3GHz.

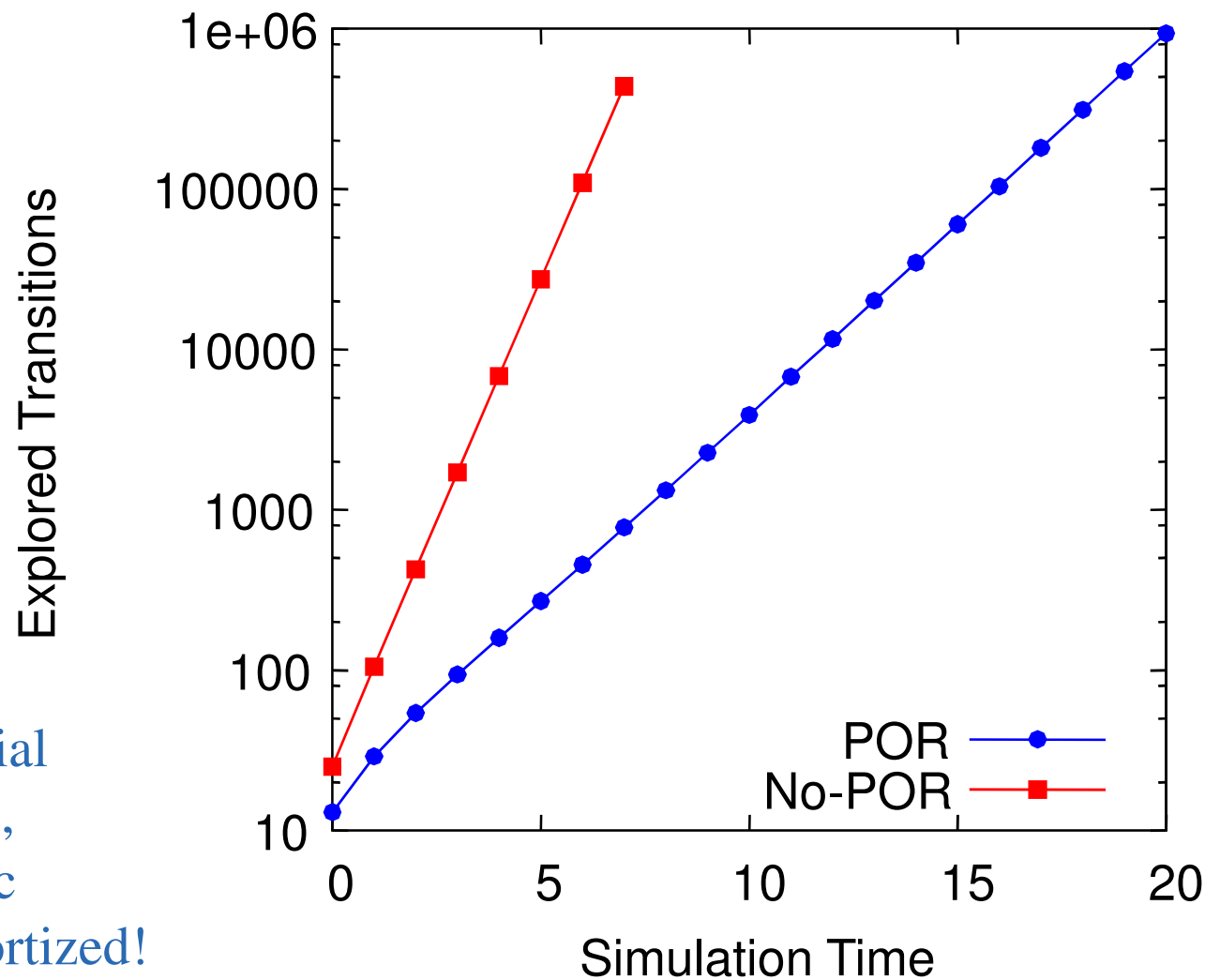
# Trading Precision for Time



Processes	Indep?	# Stren.	Predicates	SMV	Total Time
Wr0, Rd1	no	12	169	17m 6s	28m
Wr0, Rd1	no	11	42	27s	1m 30s
Wr1, Rd0	no	12	169	28m 38	39m 18s
Wr1, Rd0	no	11	42	27s	1m 30s

# Exhaustive Simulation

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.