# Compositional Approach for System Design: Semantics of SystemC

R.K. Shyamasundar
IBM Research, India Research Lab. and
Tata Institute of Fundamental Research

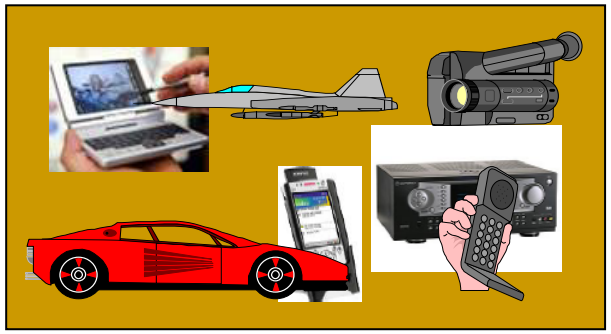
(with F. Doucet, I. Krüeger and R Gupta UCSD)

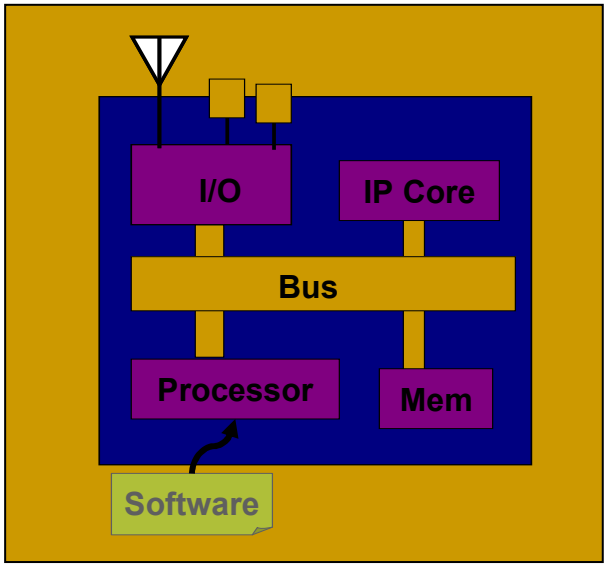
# Outline

- introduction
- compositional approach
- semantics
- anomalous behaviors
- assertional specification

# Context

- High-level modeling and design of embedded systems
  - Build “complete” system models
  - Explore potential design space
  - Iterative/exhaustive process
- Platform-based design
  - A good solution that can be customized and configured
  - provides known communication architectures



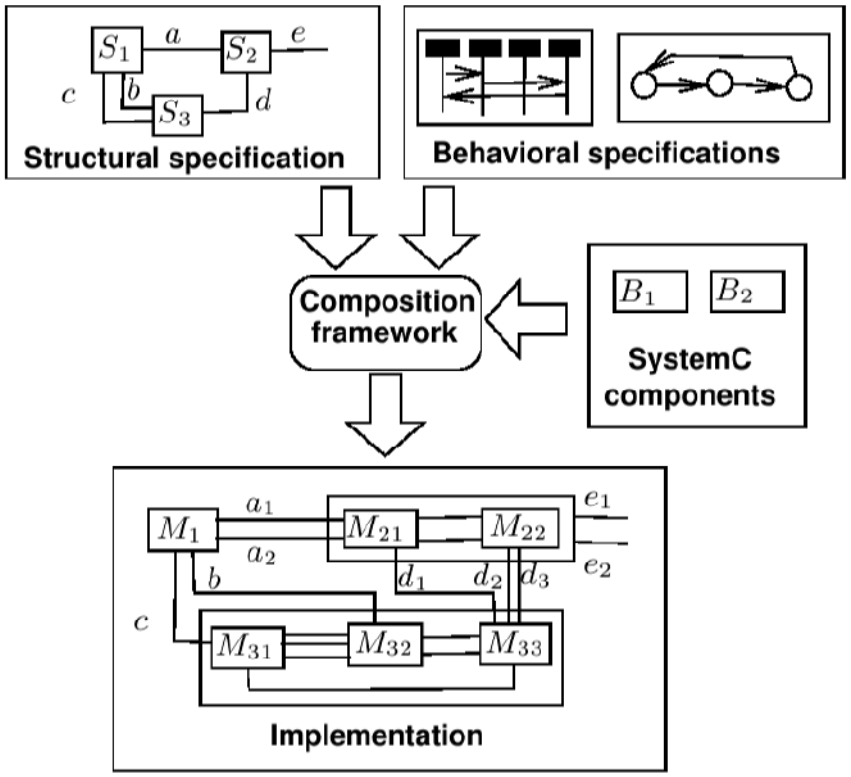


# Introduction

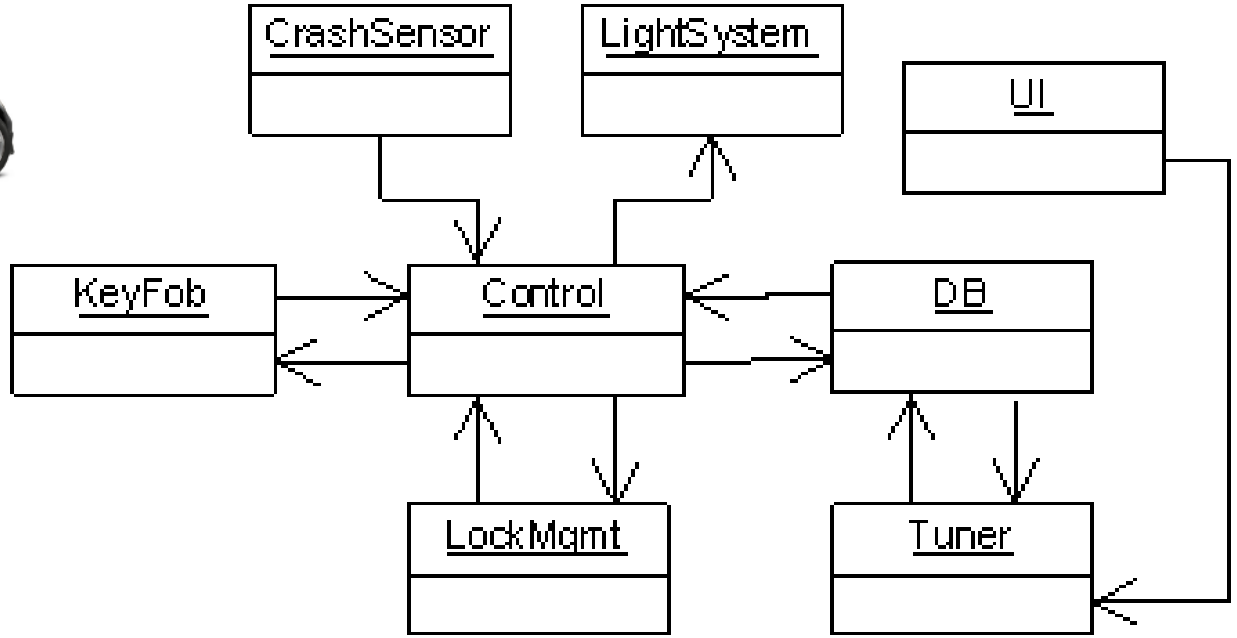
- System-level design languages (SystemC) has helped to increase the level of abstraction
- Provides a component-based design approach
  - Decompose systems into modules
- Challenge: verification of correctness of a component integration
  - Complexity of interactions
- *A compositional design methodology is necessary*
  - Analyze component by component

# Component Composition Framework

- Structural specification
  - components, channels, events, shared variables, connections
- Behavioral specification
  - *scenarios* of observable event sequences
- Components implementations
  - SystemC modules
- Incremental approach to integration and verification



# Example: Central Locking System

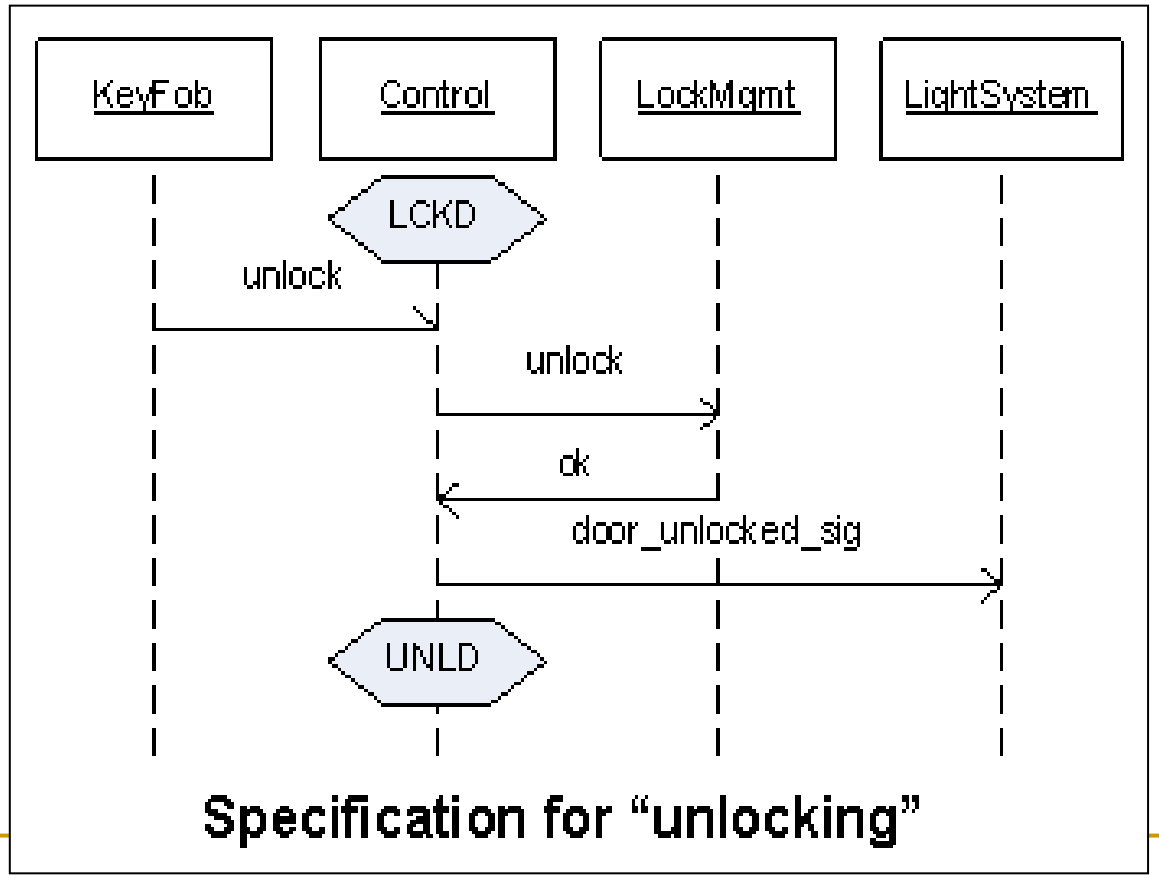



*The control system interacts with many components in the car*

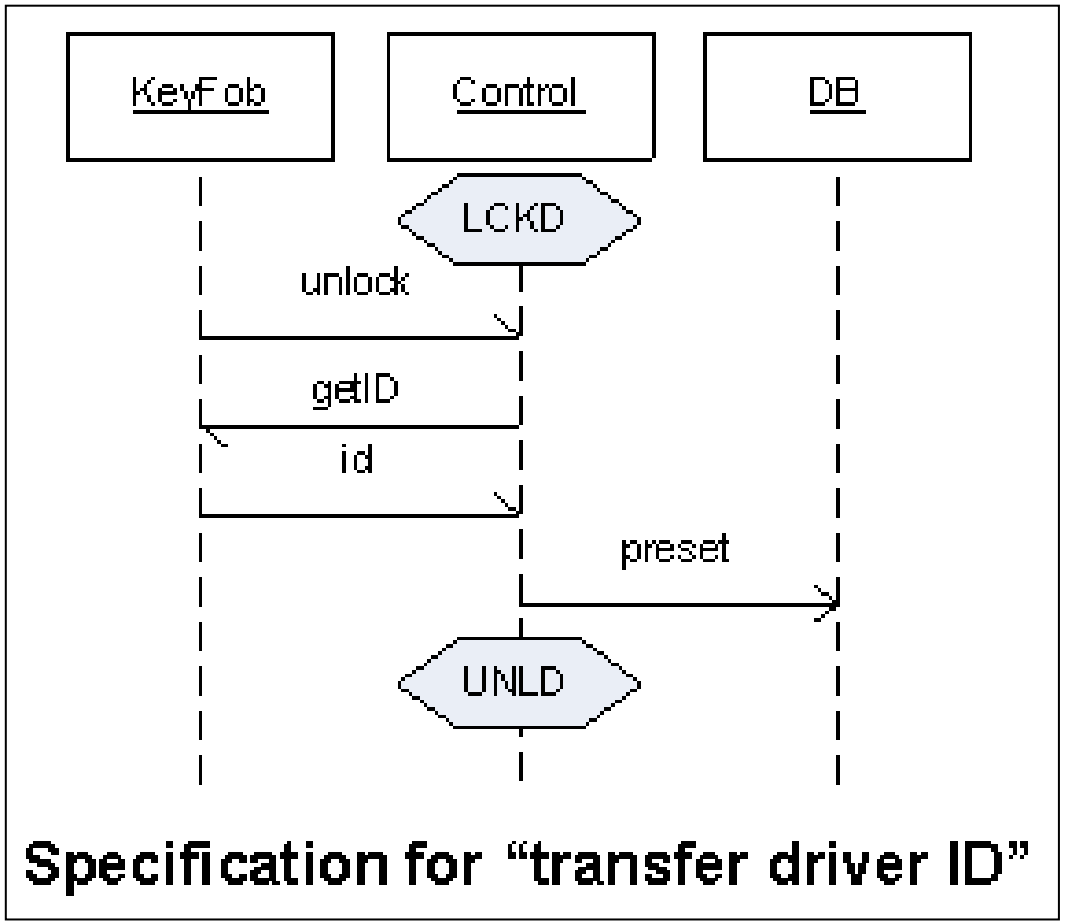
*Let us look at the specifications of interaction scenarios*

# Example: Central Locking System

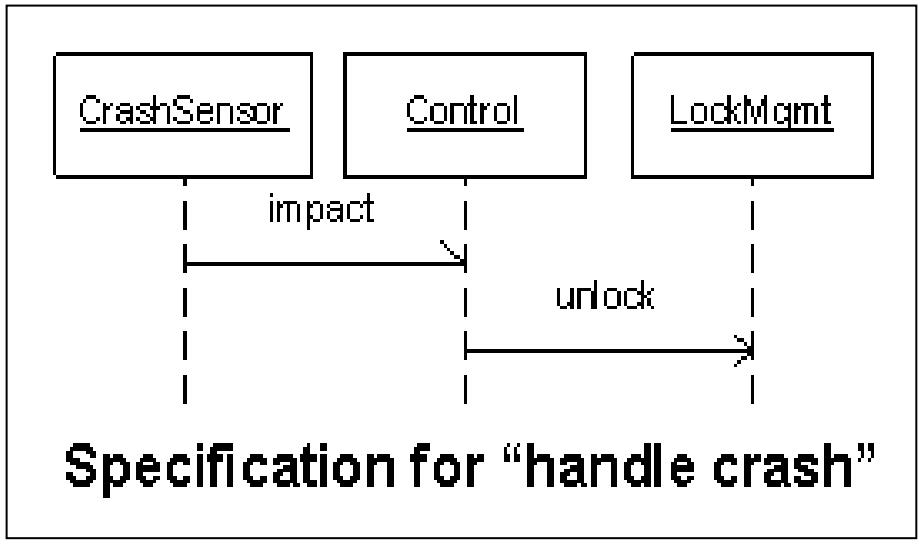
*Scenario*: observation of the interactions of many components (cross-cutting the architecture).



# Example: Central Locking System



# Example: Central Locking System

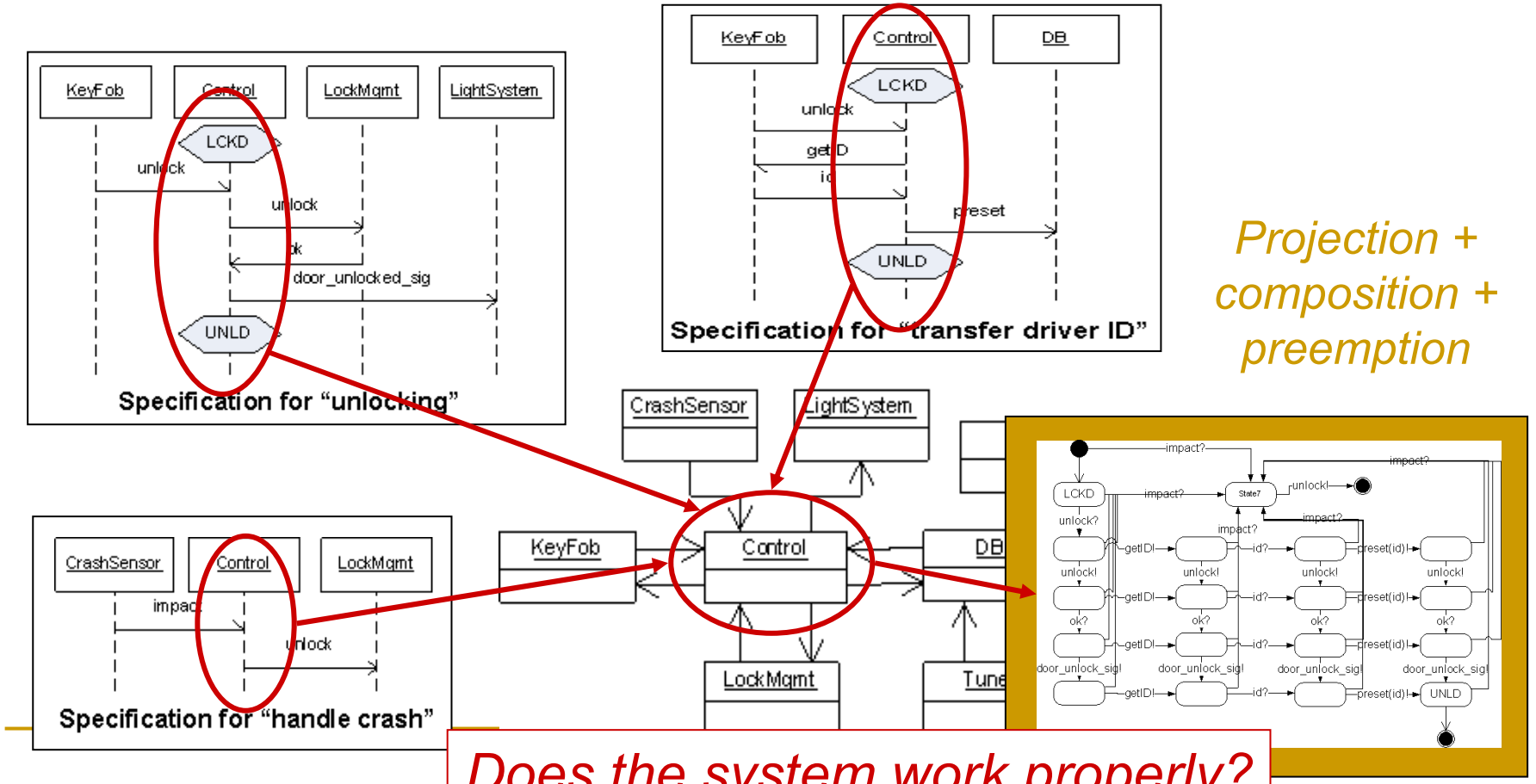



At any time of a crash, the doors should unlock!

(the specification should be verified to imply this property)

# Example: Central Locking System

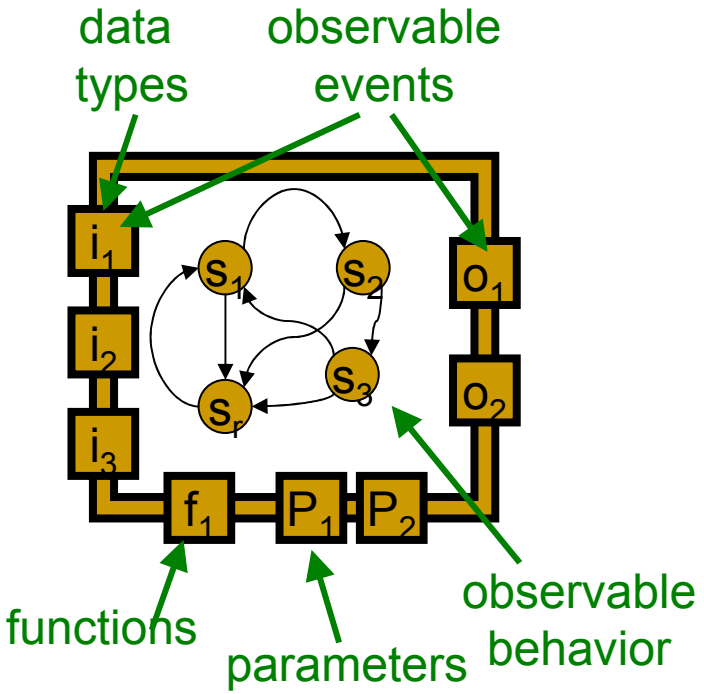
We “combine” all the scenarios to define the behavioral type of the controller



# Component Type

Let us have components types as

1. interface type
   - with classical types
2. behavioral type
   - set of all possible observable sequences at the interface




Components are SystemC blocks, or composition of SystemC blocks

Internally, component behavior is whatever but it has to *respect the "type contract"*

# How do we go about?

- Scenario: cross cutting behavior
- Component types: part which is local to a component
  - what can be observed at the interface of a component
- Semantics: clear unambiguous understanding of
  1. scenario specification
  2. IP block behaviors
- Verification of logical correctness
  - Compositional (modular) verification abstractions

# Outline

- introduction
- compositional approach
- semantics
- anomalous behaviors
- assertional specification

# SystemC Language

- C++ library to simulate concurrency
- Language: set of macros

| | |
|---|---|
| **program =** | **processes\|\| channels \|\| variables \|\| events** |
| **process** | **= ( comms \| ctrl flow \| arithmetic)*** |
| **comms =** | **event.notify() \| event.notify_delayed()<br>\| signal.read() \| signal.write() \|<br>\| channel.<transaction_name>()<br>\| port.read() \| port.write()<br>\| port.<transaction_name>()** |
| **ctrl flow =** | **if (exp) then <s1> else <s2> \| while (exp) <s1>** |

# SystemC Example

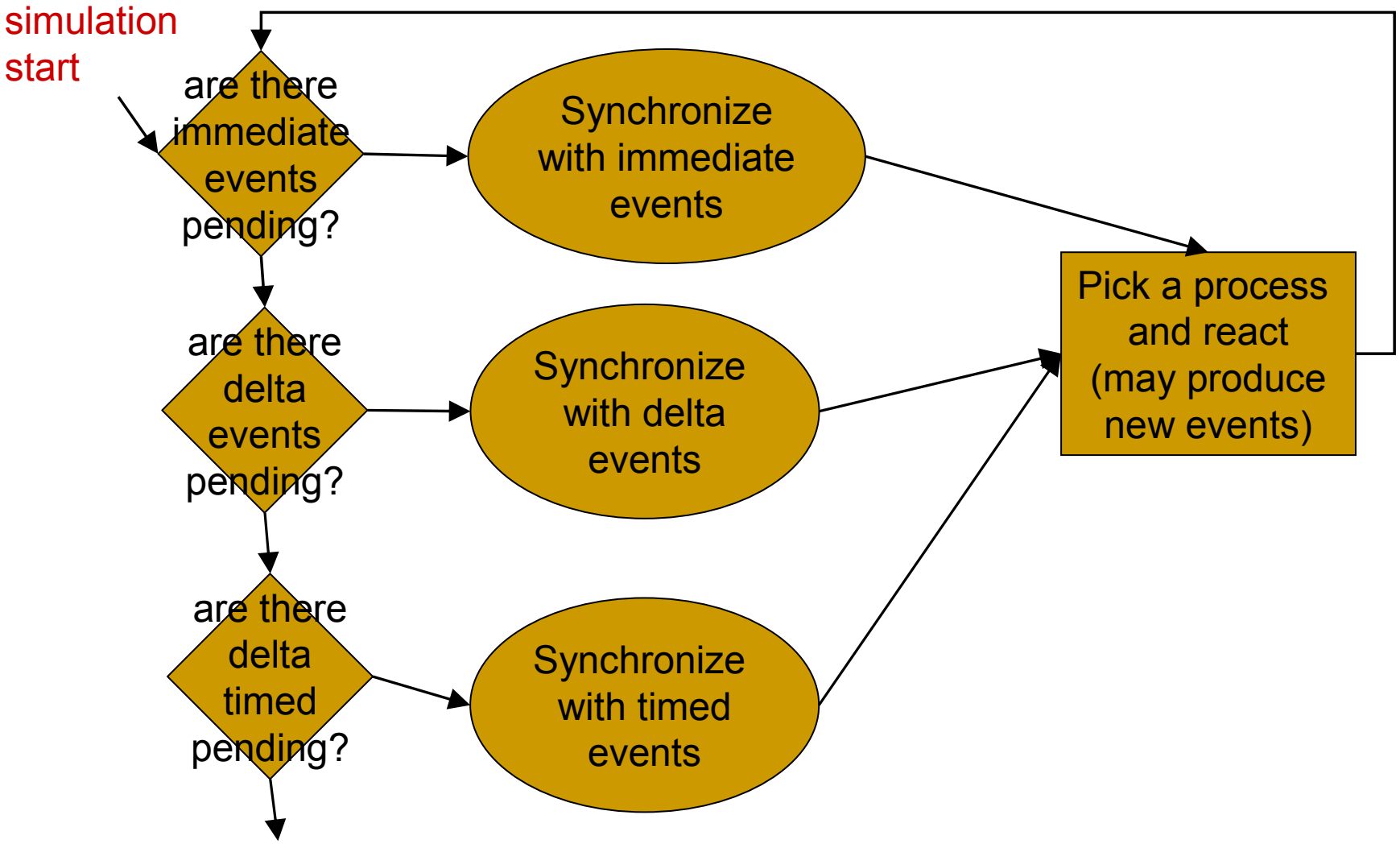
- structure:
  - module
  - ports
  - process
- communication
  - channel
  - events
  - signals

Channels and signals are bound to ports

```
SC_MODULE(Counter) {
        sc_in<bool> clk;
        sc_out<int> count;

        int value;
        SC_CTOR(Counter) {
                SC_PROCESS(proc);
                sensitive << clk;
                count=0;
        }
        void proc() {
                value++;
                count.write(value);
                wait(clk.default_event())
        }
};
```

*SystemC program is a set of concurrent modules*

# SystemC Scheduler



# Concurrency model: Two-level Timing

- Use of delta cycles to preserve deterministic behavior
  - Order events that happen within a given scheduling step
  - Event can be immediate, timed or at delta cycles
- Simulation correctness does not imply logical correctness due to
  - non-determinism
  - causality cycles

*Need to define formal semantics*

# Semantics

With a set of concurrent processes $P_1||\ldots||P_n$

a global reaction $((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$ is defined with

1. synchronize with the environment $\rightarrow_I$
2. reactive semantics (one process reacts) $\rightarrow$ asynchronous
3. build next micro-environment $\rightarrow_\delta$ synchronous
4. build next macro-environment $\rightarrow_T$ global time

which is an alternating sequence of synchronization and reactions $(\rightarrow_{sync} \rightarrow_{react})^*$, observable as a sequence of *environment* and *states* $E_0\sigma_1E_2\sigma_3E_4\sigma_5E_6\sigma_7\ldots$

# Rules for Parallel Composition

$$((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$$

**(sync-composition-immediate)**

$$\frac{\begin{array}{c} E^I \neq \emptyset \\ \forall i \in \{1..l\}, P_i \overset{S_i}{\Rightarrow}, S_i \cap E \neq \emptyset \\ \forall j \in \{l..m\}, P_j \overset{S_j}{\Rightarrow}, S_j \cap E = \emptyset \\ \forall k \in \{m..n\}, P_k \overset{r}{\Rightarrow} \end{array}}{(P_1 \| ... \| P_l \| ... \| P_m \| ... \| P_n) \xrightarrow[\langle E^I, E^\delta, L\rangle]{\langle \emptyset, E^\delta, L\rangle, 1}_I (P_1' \| ... \| P_l' \| P_{l+1} \| ... \| P_m \| ... \| P_n)}$$

Synchronize processes with the events in the environment

**(async-composition)**

$$\frac{\begin{array}{c} E^I = \emptyset \\ \forall i \in \{1..m\}, P_i \overset{S_i}{\Rightarrow} \\ \forall j \in \{m...n\}, P_j \overset{r}{\Rightarrow} \\ select\, x \in \{m..n\}, (P_x, \sigma) \xrightarrow[\langle \emptyset, E^\delta, L\rangle]{\langle E_x^I, E_{xx}^\delta, L_x\rangle, 0} (P_x' \sigma') \\ merge(E_x^I, \{E_x^\delta, E^\delta\}, \omega) \end{array}}{(P_1 \| ... \| P_m \| ... \| P_n, \sigma) \xrightarrow[\langle \emptyset, E^\delta, L\rangle]{\langle E_x^I, E_x^\delta \cup E^\delta, L_x \cup L\rangle, 1} (P_1 \| ... \| P_m \| ... \| P_x' \| ... \| P_n, \sigma')}$$

Select one process which is ready to react and fire it

# Rules for Parallel Composition

$$((\rightarrow_I \rightarrow)^* \rightarrow_\delta)^* \rightarrow_T)^*$$

**(sync-composition-micro)**

$$\frac{E^I = \emptyset \wedge E^\delta \neq \emptyset \qquad \forall i \in \{1..n\}, P_i \overset{S_i}{\Rightarrow}}{(P_1 \| ... \| P_n, \sigma) \xrightarrow[\langle \emptyset, E^\delta, \rangle]{\langle E^\delta, \emptyset, L \rangle, 1}_\delta (P_1 \| ... \| P_n, \sigma[V^\delta / V])}$$

Build next micro-environment: delta events to current events

**(sync-composition-macro)**

$$\frac{E^I = \emptyset \wedge E^\delta = \emptyset \qquad \forall i \in \{1..n\}, P_i \overset{S_i}{\Rightarrow}}{(P_1 \| ... \| P_n) \xrightarrow[\langle \emptyset, \emptyset, L \rangle]{\langle nexttime(), \emptyset, L \rangle, 1}_T (P_1 \| ... \| P_n)}$$

Build next macro-environment: timed events to current events

# Structured Operational Semantics

- For every syntactic SystemC statement *stmt*, a clean rule to derive observational behavior:

$$(stmt, \sigma) \xrightarrow[E_I]{E_O, b} (stmt', \sigma')$$

where

- $E_I$ is the triggering environment
- $E_O$ is the output environment
- $b$ denotes if the statement terminates in the current instant
- $\sigma$: denotes the state (values assigned to the variables)

- The rules are used to produce a transition system whose language is the all observable sequences

# SOS Rules for Event Communication

Produces a behavior of the form : $E_I \, \sigma \, E_O$

| | | |
|---|---|---|
| **(wait-syntactic)** | $\texttt{wait(e)} \xrightarrow[syn]{} pause; wait(e)$ | wait for the next event $e$ |
| **(pause)** | $pause \xrightarrow[E]{0} \_$ | pause until the next environment |
| **(wait-event-block)** | $\dfrac{e \notin E \wedge \neg Rv}{wait(e) \xrightarrow[E]{0} wait(e)}$ | If $e$ not in environment, wait for next instant |
| **(wait-event-unblock)** | $\dfrac{e \in E \wedge \neg Rv}{wait(e) \xrightarrow[E]{1} \_}$ | If $e$ is in the environment, reduction terminates |
| **(event-notify)** | $\texttt{e.notify()} \xrightarrow[E]{e,\emptyset,\emptyset,1} \_$ | $e$ in the next environment |

# SOS Rules for Sequential Composition

Produces a behavior of the form: $E_1\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5E_O$

**(assignement)** $(\mathtt{x:=e}, \sigma) \xrightarrow[E]{1} (\_, \sigma'[x/e])$ Changes the state $\sigma$ into $\sigma'$

**(sequential-composition-1)**

$$\frac{(P_1, \sigma) \xrightarrow[E]{E_O, E_O^\delta, L_1, 0} (P_1', \sigma_1')}{(P_1; P_2, \sigma) \xrightarrow[E]{E_O, E_O^\delta, L_1, 0} (P1'; P_2, \sigma_1')}$$

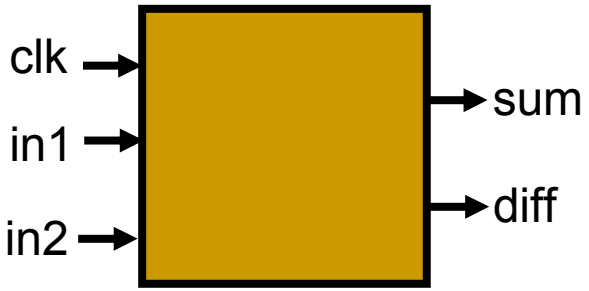
$P_1$ blocks, the whole sequential composition blocks

**(sequential-composition-2)**

$$\frac{(P_1, \sigma) \xrightarrow[E]{E_{O_1}, E_{O_1}^\delta, L_1, 1} (P_1', \sigma_1') \qquad (P_2, \sigma_1') \xrightarrow[E]{E_{O_2}, E_{O_2}^\delta, L_2, b_2} (P_2', \sigma_2') \qquad \mathtt{merge}(\langle E_{O_1}, E_{O_2}\rangle, \langle E_{O_1}, E_{O_2}\rangle, \omega)}{(P_1; P_2, \sigma) \xrightarrow[E]{E_{O_1} \cup E_{O_2}, E_{O_1}^\delta \cup E_{O_2}^\delta, L_1 \cup L_2, b_2} (P_2', \sigma_2')}$$

# Example

Module definition

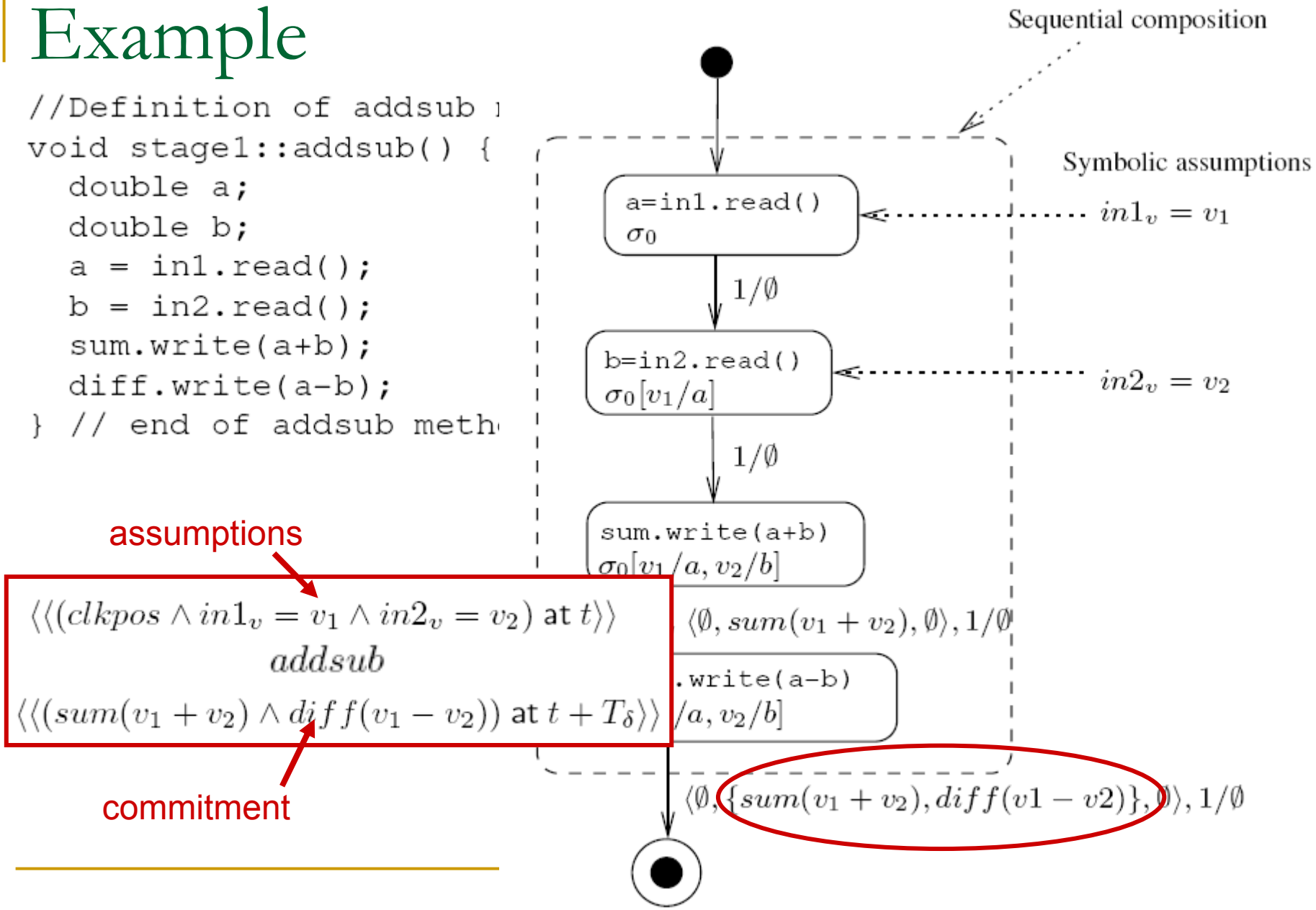



Process definition

```
struct stage1 : sc_module {
    sc_in<double> in1;    //input 1
    sc_in<double> in2;    //input 2
    sc_out<double> sum;   //output 1
    sc_out<double> diff; //output 2
    sc_in<bool>    clk;   //clock
    void addsub();
    //Counstructor
    SC_CTOR( stage1 ) {
        //Declare addsub as SC_METHOD and
        SC_METHOD( addsub );
        dont_initialize();
        // make it sensitive to positive clo
        sensitive_pos << clk;
    }
};
//Definition of addsub method
void stage1::addsub() {
  double a;
  double b;
  a = in1.read();
  b = in2.read();
  sum.write(a+b);
  diff.write(a-b);
} // end of addsub method
```

# Example

```
//Definition of addsub
void stage1::addsub() {
  double a;
  double b;
  a = in1.read();
  b = in2.read();
  sum.write(a+b);
  diff.write(a-b);
} // end of addsub meth
```

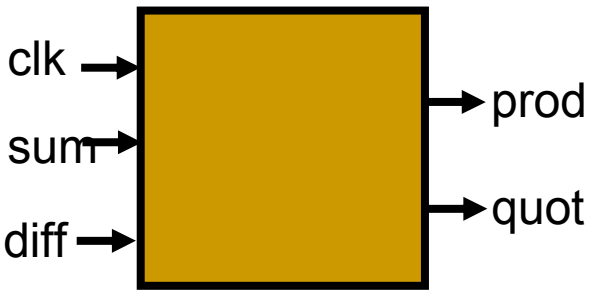


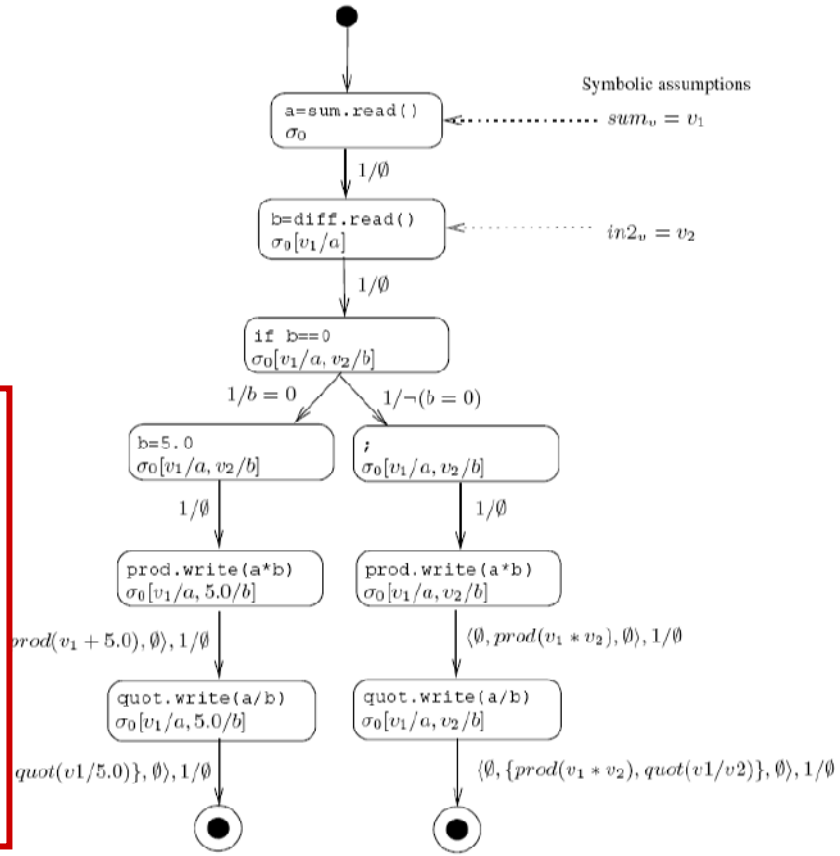
# Example:

```
//definition of multdiv method
void stage2::multdiv() {
  double a;
  double b;
  a = sum.read();
  b = diff.read();
  if( b == 0 )
      b = 5.0;
  prod.write(a*b);
  quot.write(a/b);
} // end of multdiv
```

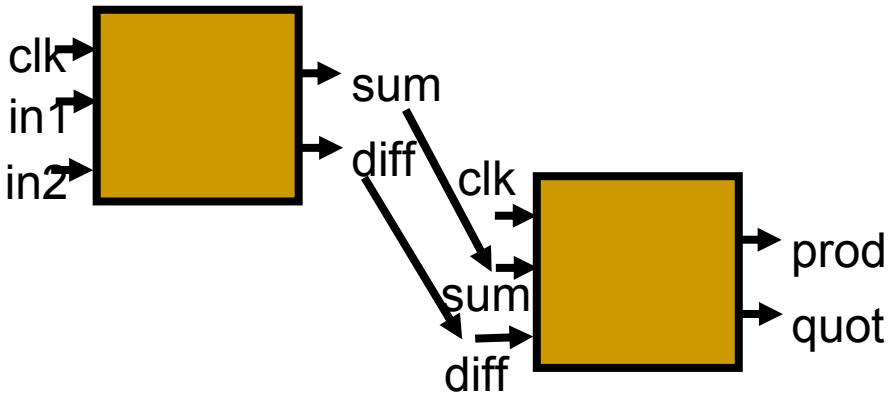
$$\langle\langle (clkpos \wedge sum_v = v_1 \wedge diff_v = v2) \text{ at } t \rangle\rangle$$
$$multdiv$$
$$\langle\langle (prod(v_1 * v_2) \wedge quot(v_1/v_2)) \text{ at } t + T_\delta \rangle\rangle$$

$$\langle\langle (clkpos \wedge sum_v = v_1 \wedge diff_v = 0) \text{ at } t \rangle\rangle$$
$$multdiv$$
$$\langle\langle (prod(v_1 * 5.0) \wedge quot(v_1/5.0)) \text{ at } t + T_\delta \rangle\rangle$$



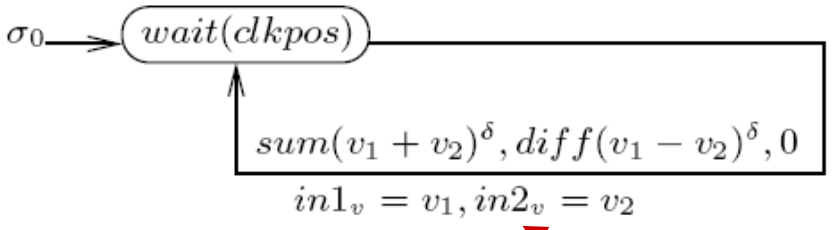



# Example




Automata collaped with sequential rule and with process loop:

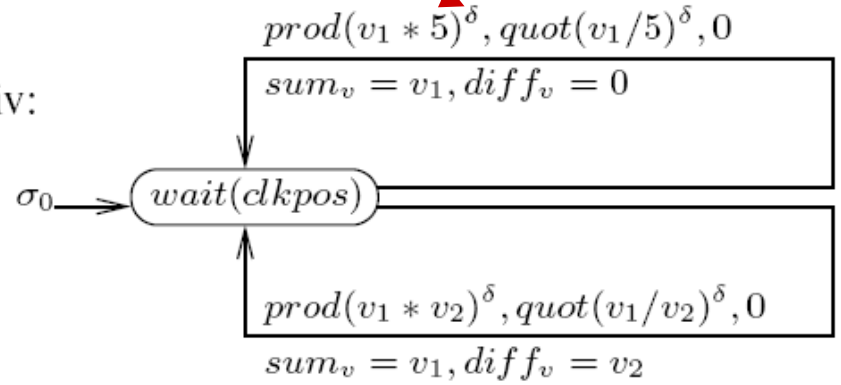
addsub:




assumptions

delta events

multdiv:




Assemble the interface behaviors of the SystemC blocks:

$$\langle\langle (in1 = v_1 \wedge in2 = v_2 \wedge clkpos \wedge sum_v = v_1 \wedge diff_v = v2) \text{ at } t\rangle\rangle$$
$$addsub \| multdiv$$
$$\langle\langle (sum(v_1+v_2) \wedge diff(v_1-v_2)) \text{ at } t + T_\delta \wedge$$
$$((prod(v_3 * v_4) \wedge quot(v_3/v_4)) \vee (prod(v_3 * 5) \wedge quot(v_3/5)) \text{ at } t + T_\delta\rangle\rangle$$

# Outline

- introduction
- compositional approach
- semantics
- anomalous behaviors
  - non-determinism
  - causality cycles
- assertional specification

# Nondeterministic Behavior

- For an input trace, it can be possible to observe different output traces
  - can cause synchronization problems
  - missed events, different values, etc
- Possible causes:
  1. mix of concurrency with shared variables
  2. mix of concurrency with immediate event notification
  3. non-deterministic software models with immediate event notifications
  4. un-initialized signals/variables
- Scheduler-dependent behavior
  - not observable in a simulation model

# Nondeterministic Behavior

event notification can be missed depending of which process gets scheduled first

```
SC_MODULE(M1) {
  sc_event e;
  int data;

  SC_CTOR(M) {
    SC_THREAD(a);
    SC_THREAD(b);
  }
  void a() {
    data=1;
    e.notify()
  }
  void b() {
     wait(e)
  }
};
```

at the initial step

```
SC_MODULE(M2) {
  sc_event e;

  SC_CTOR(M) {
    SC_THREAD(a);
    SC_THREAD(b);
  }
  void a() {
    wait(10,SC_NS)
    e.notify();
  }
  void b() {
    wait(10, SC_NS);
    wait(e);
  }
};
```

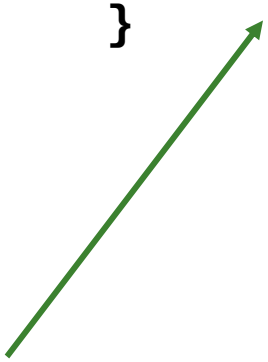
at some arbitrary step

# Scheduler Dependency

```
sc_event e;

SC_MODULE(M1) {

  SC_CTOR(M1) {
    SC_THREAD(a);
  }
  void a() {
    e.notify()
  }
};
```

```
SC_MODULE(M2) {

  SC_CTOR(M2) {
    SC_THREAD(b);
  }
  void b() {
    wait(e);
    sc_stop();
  }
};
```

```
int sc_main() {
  M1 m1(''m1'');
  M2 m2(''m2'');

  sc_start(10);
  return 1;
}
```



This runs to completion and execute the sc_stop statement

# Scheduler Dependency

```
sc_event e;

SC_MODULE(M1) {

  SC_CTOR(M1) {
    SC_THREAD(a);
  }
  void a() {
    e.notify()
  }
};
```

```
SC_MODULE(M2) {

  SC_CTOR(M2) {
    SC_THREAD(b);
  }
  void b() {
    wait(e);
    sc_stop();
  }
};
```

```
int sc_main() {
  M1 m1(''m1'');
  M2 m2(''m2'');

  sc_start(10);
  return 1;
}
```

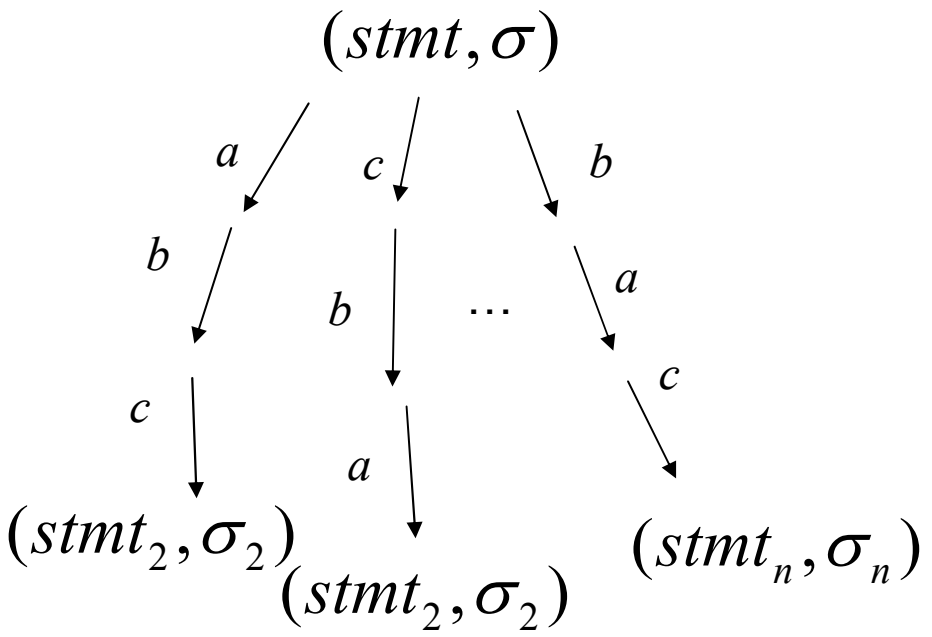
```
int sc_main() {
 M2 m2(''m2'');
 M1 m1(''m1'');

  sc_start(10);
  return 1;
}
```

inverting the instantiation order makes M2 miss e and block forever

**Structural specification has side effects!!!**

# Detecting nondeterminism

- Has to be done during the synchronous composition
  - merge for delta events
  - merge for timed events
- Keep track of the
  - last environment $E$
  - last state σ
  - check if all derivations lead to
    - same environment $E'$
    - same state σ'




$$(stmt_1 = stmt_2 = \ldots = stmt_n) \wedge (\sigma_1 = \sigma_2 = \ldots = \sigma_n)$$

(like an interference freedom test)

# Causal Cycles

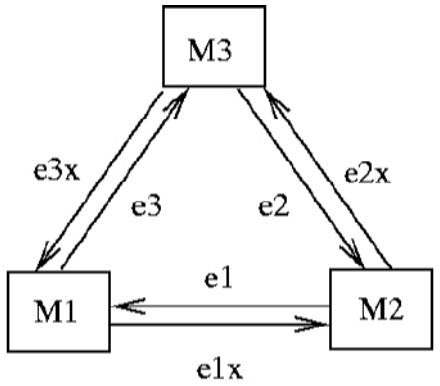
- A sequence of action that keeps retriggering itself
- Problem with delta timing :
  - infinite actions in a finite time

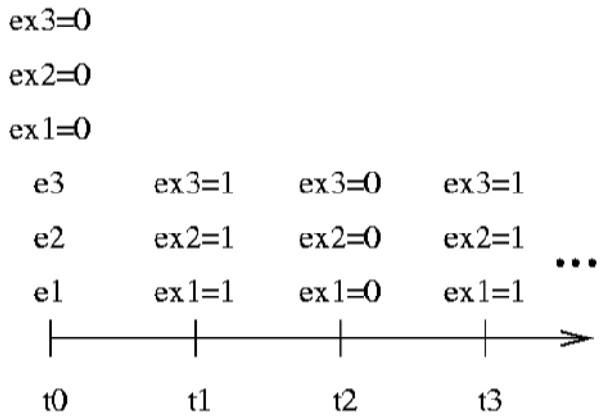
- Not always a problem
  - desirable in an untimed model
  - undesirable in a timed model (considered as a divergence)
- Cycles are not always triggered in simulation
  - could be a corner case condition of asynchronous logic
  - need to detect in verification

# Example:

## Checking for absence of event forms a cycle

```
SC_MODULE(M1) {
  sc_in<bool>  e1;
  sc_in<bool>  e3x;
  sc_out<bool> e3;
  sc_out<bool> e1x;

  SC_CTOR(M1) {
    SC_METHOD(p1);
    sensitive << e1 << e3x;
  }
  void p1() {
    if (!e3x.event())
      e3.write(!e3.read());
    e1x.write(!e1x.read());
  }
};
```




(b)

# Detecting causal cycles

- Makes sense only in timed models
  - using global explicit time
- A derivation that never gets to build the next timed environment
  - a cycle in the transition graph where no T transition is taken.

This step never taken

$$\left(\left(\rightarrow_I \rightarrow\right)^* \rightarrow_\delta\right)^* \rightarrow_T)^*$$

  - Can be done while constructing the graph if it goes back to a state already in the graph

# Current work on abstraction

- Abstraction of TLM models
  - abstract transactions as syntactic statements

$$(trans,\sigma)\xrightarrow[E_1]{E_2}\ldots\xrightarrow[E_{n-1}]{E_n}(\_,\sigma') \approx (trans,\sigma)\xrightarrow[E_1]{E_n}(\_,\sigma')$$

  - establish interference freedom and cooperation tests
    - transaction should not interfere with each other
    - transaction should be abstracted as asynchronous tasks
- Compositional methodology
  1. verification of bus
  2. verification of components
  3. deduction of system properties

# Related Work

Semantics of SystemC: difficult problem because of all its intricacies of two-level model

- ASML (Tahar/Mueler …)
  - integration of time and delta cycle models
  - difficult integration of asynchrony
  - no reactivity (deep embedding into ASML environment)
- Synchronous languages
  - formalization through Lustre (Maraninchi, Moy …)
    - ignores delta cycle requirements
  - formalization through SIGNAL (Talpin)
    - ignores delta cycle
    - difficult integration with asynchrony
- Process algebras (Kam …)
  - ignores distinction of synchrony/asynchrony
- Verification approaches (Kroening …)
  - not geared towards arriving at simulation correctness

# Summary and Ongoing Work

- A Framework for component composition
- Defined compositional semantics as supported in SystemC for enabling a component-based design approach
- Currently defining assertional spec. layer
  - Generation of automata (TS) for model checking
  - accompanying verification methodology for model checking
- Apply on TS, Predicate abstractions, solvers, for scalable verification …