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
System-level design languages (SystemC) has helped to increase the level of abstraction. Provides a component-based design approach. 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

Specification for “transfer driver ID”
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.

Projection + composition + preemption

Does the system work properly?
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

\[
\begin{align*}
\text{program} &= \text{processes} \parallel \text{channels} \parallel \text{variables} \parallel \text{events} \\
\text{process} &= (\text{comms} \mid \text{ctrl flow} \mid \text{arithmetic})^* \\
\text{comms} &= \text{event.notify()} \mid \text{event.notify\_delayed()} \\
&\quad \mid \text{signal.read()} \mid \text{signal.write()} \\
&\quad \mid \text{channel.<transaction\_name>()} \\
&\quad \mid \text{port.read()} \mid \text{port.write()} \\
&\quad \mid \text{port.<transaction\_name>()} \\
\text{ctrl flow} &= \text{if (exp) then } <s1> \text{ else } <s2> \mid \text{while (exp) } <s1>
\end{align*}
\]
SystemC Example

- **structure:**
  - module
  - ports
  - process

- **communication**
  - channel
  - events
  - signals

Channels and signals are bound to ports

SystemC program is a set of concurrent modules

```c
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 Scheduler

Simulation start

- Are there immediate events pending?
- Are there delta events pending?
- Are there delta timed events pending?

Synchronize with immediate events
Synchronize with delta events
Synchronize with timed events

Pick a process and react (may produce new events)

Simulation done
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 $P_1||...||P_n$

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 $E_0\sigma_1E_2\sigma_3E_4\sigma_5E_6\sigma_7...$
Rules for Parallel Composition

(sync-composition-immediate)

\[ E^I \neq \emptyset \]
\[ \forall i \in \{1..l\}, P_i \xrightarrow{S_i}, S_i \cap E \neq \emptyset \]
\[ \forall j \in \{l..m\}, P_j \xrightarrow{S_j}, S_j \cap E = \emptyset \]
\[ \forall k \in \{m..n\}, P_k \xrightarrow{r} \]

\[ (P_1 || \ldots || P_l || \ldots || P_m || \ldots || P_n) \xrightarrow{\langle \emptyset, E^\delta, L \rangle, 1} (P'_1 || \ldots || P'_l || P_{l+1} || \ldots || P_m || \ldots || P_n) \]

(Async-composition)

\[ E^I = \emptyset \]
\[ \forall i \in \{1..m\}, P_i \xrightarrow{S_i} \]
\[ \forall j \in \{m..n\}, P_j \xrightarrow{r} \]

select \( x \in \{m..n\}, (P_x, \sigma) \xrightarrow{\langle E^I_x, E^\delta_x, L_x \rangle, 0} (P'_x, \sigma') \)

merge(\( E^I_x, \{ E^\delta_x, E^\delta \}, \omega \))

\[ (P_1 || \ldots || P_m || \ldots || P_n, \sigma) \xrightarrow{\langle E^I_x, E^\delta_x \cup E^\delta, L_x \cup L \rangle, 1} (P'_1 || \ldots || P'_m || \ldots || P'_x || \ldots || P_n, \sigma') \]
Rules for Parallel Composition

\[ ((\rightarrow I \rightarrow)^* \rightarrow \delta)^* \rightarrow T)^* \]

**Sync-composition-micro**

\[ \begin{align*}
    E^I &= \emptyset \land E^\delta \neq \emptyset \\
    \forall i \in \{1\ldots n\}, &\ P_i \xrightarrow{S_i} \\
    (P_1 \parallel \ldots \parallel P_n, \sigma) \xrightarrow{\langle E^\delta, \emptyset, L \rangle, 1}_{\delta} &\ (P_1 \parallel \ldots \parallel P_n, \sigma[V^\delta / V])
\end{align*} \]

Build next micro-environment: delta events to current events

**Sync-composition-macro**

\[ \begin{align*}
    E^I &= \emptyset \land E^\delta = \emptyset \\
    \forall i \in \{1\ldots n\}, &\ P_i \xrightarrow{S_i} \\
    (P_1 \parallel \ldots \parallel P_n) \xrightarrow{\langle \text{nexttime}(), \emptyset, L \rangle, 1}_{T} &\ (P_1 \parallel \ldots \parallel P_n)
\end{align*} \]

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:

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

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: $E_I \sigma E_O$

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(wait-syntactic)</td>
<td>$\text{wait}(e) \xrightarrow{\text{syn}} pause; \text{wait}(e)$</td>
<td>wait for the next event $e$</td>
</tr>
<tr>
<td>(pause)</td>
<td>$pause \xrightarrow{0} E$</td>
<td>pause until the next environment</td>
</tr>
<tr>
<td>(wait-event-block)</td>
<td>$e \notin E \land \neg Rv \quad \xrightarrow{\text{wait}(e)} \xrightarrow{0} \text{wait}(e)$</td>
<td>If $e$ not in environment, wait for next instant</td>
</tr>
<tr>
<td>(wait-event-unblock)</td>
<td>$e \in E \land \neg Rv \quad \xrightarrow{\text{wait}(e)} 1 \xrightarrow{E} _-$</td>
<td>If $e$ is in the environment, reduction terminates</td>
</tr>
<tr>
<td>(event-notify)</td>
<td>$e.\text{notify()} \xrightarrow{e,\emptyset,\emptyset,1} E$</td>
<td>$e$ in the next environment</td>
</tr>
</tbody>
</table>
## SOS Rules for Sequential Composition

Produce a behavior of the form: \( E_1 \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 E_0 \)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(assignement)</strong></td>
<td>( (x := e, \sigma) \xrightarrow{E} (_, \sigma'[x/e]) ) Changes the state ( \sigma ) into ( \sigma' ).</td>
</tr>
<tr>
<td><strong>(sequential-composition-1)</strong></td>
<td>( (P_1, \sigma) \xrightarrow{E_{O_1}, E_{O_1}^\delta, L_1, 0} (P_1', \sigma_1') ) ( (P_1; P_2, \sigma) \xrightarrow{E_{O_1}, E_{O_2}^\delta, L_1, 0} (P_1'; P_2, \sigma_1') )</td>
</tr>
<tr>
<td><strong>(sequential-composition-2)</strong></td>
<td>( (P_1, \sigma) \xrightarrow{E_{O_1}, E_{O_1}^\delta, L_1, 1} (P_1', \sigma_1') ) ( (P_2, \sigma_1') \xrightarrow{E_{O_2}, E_{O_2}^\delta, L_2, b_2} (P_2', \sigma_2') ) ( \text{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_{O_1 \cup O_2}, E_{O_1 \cup O_2}^\delta, L_1 \cup L_2, b_2} (P_2', \sigma_2') )</td>
</tr>
</tbody>
</table>
```c
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(),
    //Constructor
    SC_CTOR( stage1 ) {
        //Declare addsub as SC_METHOD and
        SC_METHOD( addsub );
        dont_initialize();
        // make it sensitive to positive clock
        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

\[
\langle\langle\langle clkpos \land sum_v = v_1 \land diff_v = v_2 \rangle\rangle\rangle t
\]

\[
\langle\langle prod(v_1 \times v_2) \land quot(v_1/v_2) \rangle\rangle t + T_\delta
\]

\[
\langle\langle clkpos \land sum_v = v_1 \land diff_v = 0 \rangle\rangle t
\]

\[
\langle\langle prod(v_1 \times 5.0) \land quot(v_1/5.0) \rangle\rangle t + T_\delta
\]
Example

Assemble the interface behaviors of the SystemC blocks:

\[
\langle\langle (in1 = v_1 \land in2 = v_2 \land clkpos \land sum_v = v_1 \land diff_v = v_2) \text{ at } t\rangle\rangle \\
\text{addsub} || \text{multdiv} \\
\langle\langle (sum(v_1 + v_2) \land diff(v_1 - v_2)) \text{ at } t + T_\delta \land \\
((prod(v_3 * v_4) \land quot(v_3/v_4)) \lor (prod(v_3 * 5) \land quot(v_3/5)) \text{ at } t + T_\delta)\rangle\rangle
\]
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 on which process gets scheduled first.

```c
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);
    }
};
```

```c
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 the initial step

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

```c
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;
}
```

inverting the instantiation order makes M2 miss e and block forever

```c
int sc_main() {
  M2 m2('m2''');
  M1 m1('m1''');
  sc_start(10);
  return 1;
}
```

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 $\sigma$
  - check if all derivations lead to
    - same environment $E'$
    - same state $\sigma'$

\[(stmt,\sigma)\]

\[\begin{align*}
& (stmt_1, \sigma_1) \\
& (stmt_2, \sigma_2) \\
& (stmt_3, \sigma_3) \\
& \ldots \\
& (stmt_n, \sigma_n)
\end{align*}\]

\[(stmt_1 = stmt_2 = \ldots = stmt_n) \land (\sigma_1 = \sigma_2 = \ldots = \sigma_n)\]

(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

```cpp
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());
    }
};
```

Checking for absence of event forms a cycle

![Diagram showing the relationships between M1, M2, and M3 with events e1, e2, e3, and e1x, e3x.](image)

- ex3=0
- ex2=0
- ex1=0
- e3 ex3=1 ex3=0 ex3=1
- e2 ex2=1 ex2=0 ex2=1
- e1 ex1=1 ex1=0 ex1=1

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

\[
(\left( \rightarrow I \rightarrow \right)^* \rightarrow \delta)^* \rightarrow T)^* 
\]
Current work on abstraction

- Abstraction of TLM models
  - abstract transactions as syntactic statements
    \[
    (\text{trans}, \sigma) \xrightarrow{E_1} \ldots \xrightarrow{E_n} (_, \sigma') \approx (\text{trans}, \sigma) \xrightarrow{E_1} (_, \sigma')
    \]
  - 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 …