An Introduction to SystemC 1.0.x

Bernhard Niemann
Fraunhofer Institute for Integrated Circuits
Objectives

This tutorial offers
- A short motivation for C based design flow
- A brief introduction to the most important SystemC 1.0.x language elements
- A very short overview on refinement for synthesis
- A simple example program

This tutorial does not
- provide a complete treatment of the SystemC language
- cover system level modeling with SystemC 2.0
<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SystemC Introduction</td>
</tr>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>3</td>
<td>SystemC and Synthesis</td>
</tr>
<tr>
<td>4</td>
<td>Simple Example</td>
</tr>
<tr>
<td>5</td>
<td>Resources</td>
</tr>
</tbody>
</table>
4. Hand over specification document

Problems: Written specifications are incomplete and inconsistent
Translation to HDL is **time consuming** and **error prone**
C++ Based Flow

1. Conceptualize
2. Simulate in C++
3. Write specification document

4. Hand over
   - Executable specification
   - Testbench
   - Written specification

5. Understand
6. Refine in C++
7. Verify reusing testbenches
8. Synthesize from C++
Can C++ be used as is?

- **SystemC supports**
  - *Hardware style communication*
    - Signals, protocols, etc.
  - *Notion of time*
    - Time sequenced operations.
  - *Concurrency*
    - Hardware and systems are inherently concurrent, i.e. they operate in parallel.
  - *Reactivity*
    - Hardware is inherently reactive, it responds to stimuli and is in constant interaction with its environment, which requires handling of exceptions.
  - *Hardware data types*
    - Bit type, bit-vector type, multi-valued logic type, signed and unsigned integer types and fixed-point types.
  - *Integrated Simulation Kernel*

- All these features are not supported by C++ as is
SystemC Design Flow

Software

UTF

Hardware

TF

Partition

Behavioral Level
- synthesizable
- clocked
- I/O cycle accurate
- algorithmic description

System/Architectural Level
- not synthesizable
- event driven
- abstract communication
- abstract data types

Behavioral

Refine

Behavioral

RTL

Refine

Netlist

Refine

target code

abstract RTOS

Refine

Software Hardware
Developing SystemC - An Overview

class library and simulation kernel

your standard C/C++ development environment

header files

libraries

compiler

linker

debugger

make

Executable = simulator

source files for system and testbenches

header files

DSP

Interface

IP-Core

ASIC

© FhG IIS-A, 2001
Model Structure - System Level

Channel (implements I/F) → Interface
Module

Port (accesses a channel’s I/F)

Algorithm
Process (reads data from input ports and writes data to output ports)

SystemC 2.x
- Module
- Port
- Interface
- Channel
- Process

© FhG IIS-A, 2001
Model Structure - Implementation Level

Hardware signal

Special ports to access the hardware signal

Module

Algorithm

Process (reads data from input ports and writes data to output ports)

SystemC 1.0.x
- Module
- Special Ports
- Hardware Signal
- Process

© FhG IIS-A, 2001
## Comparison SystemC 1.0.x - VHDL

<table>
<thead>
<tr>
<th></th>
<th>VHDL</th>
<th>SystemC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hierarchy</strong></td>
<td>entity</td>
<td>module</td>
</tr>
<tr>
<td><strong>Connection</strong></td>
<td>port</td>
<td>port</td>
</tr>
<tr>
<td><strong>Communication</strong></td>
<td>signal</td>
<td>signal</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>process</td>
<td>process</td>
</tr>
<tr>
<td><strong>Test Bench</strong></td>
<td></td>
<td>object orientation</td>
</tr>
<tr>
<td><strong>System Level</strong></td>
<td></td>
<td>channel, interface, event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>abstract data types</td>
</tr>
<tr>
<td><strong>I/O</strong></td>
<td>simple file I/O</td>
<td>C++ I/O capabilities</td>
</tr>
<tr>
<td>Unit</td>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SystemC Introduction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SystemC and Synthesis</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Simple Example</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Resources</td>
<td></td>
</tr>
<tr>
<td>Unit</td>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Language Elements of SystemC</strong></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td><strong>Data Types</strong></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td><strong>Modules, Ports and Signals</strong></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td><strong>Processes</strong></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td><strong>Hierarchy</strong></td>
<td></td>
</tr>
</tbody>
</table>
C++ built in data types may be used but are not adequate to model Hardware.

- long, int, short, char, unsigned long, unsigned int, unsigned short, unsigned char, float, double, long double, and bool.

SystemC™ provides other types that are needed for System modeling.

- Scalar boolean types: sc_logic, sc_bit
- Vector boolean types: sc_bv<length>, sc_lv<length>
- Integer types: sc_int<length>, sc_uint<length>, sc_bigint<length>, sc_biguint<length>
- Fixed point types: sc_fixed, sc_ufixed
To get fast simulations, you have to choose the right data type

- use builtin C/C++ data types as much as possible
- use sc_int/sc_uint
  - integer with up to 64 bits
  - model arithmetic and logic operations
- use sc_bit/sc_bv
  - arbitrary length bit vector
  - model logic operations on bit vectors
  - two valued logic
- use sc_logic/sc_lv
  - 4 valued logic vectors
  - model tri-state behavior

*Hint:* You may use a profiling tool, which tells you, how much time you spent in which function call, to control simulation speed.
To get fast simulations, you have to choose the right data type

- **Fastest**
  - use `sc_bigint/sc_biquint`
    - integer with arbitrary length
    - use to model arithmetic operations on large bit vectors
    - ineffective for logic operations
  - use `sc_fixed/sc_ufixed`
    - arbitrary precision fixed point
    - use to model fixed point arithmetic
    - ineffective for logic operations

**Hint:** You may use a profiling tool, which tells you, how much time you spent in which function call, to control simulation speed.
**sc_int, sc_uint, sc_bigint, sc_biguint Syntax**

Syntax:

```cpp
sc_int<length> variable_name;
sc_uint<length> variable_name;
sc_bigint<length> variable_name;
sc_biguint<length> variable_name;
```

**length:**
- Specifies the number of elements in the array
- Must be greater than 0
- Must be compile time constant
- Use `[]` to bit select and `range()` to part select.
- Rightmost is LSB(0), Leftmost is MSB (n-1)

```cpp
sc_int<5> a; // a is a 5-bit signed integer
sc_uint<44> b; // b is a 44-bit unsigned integer
sc_bigint<5> c;
c = 13; // c gets 01101, c[4] = 0, c[3] = 1, …, a[0] = 1
bool d;
d = c[4]; // d gets 0
d = c[3]; // d gets 1
sc_biguint<3> e;
e = c.range(3, 1); // e gets 110 - interpreted as -2
```
<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>2.1</td>
<td>Data Types</td>
</tr>
<tr>
<td>2.1</td>
<td>Modules, Ports and Signals</td>
</tr>
<tr>
<td>2.2</td>
<td>Processes</td>
</tr>
<tr>
<td>2.3</td>
<td>Hierarchy</td>
</tr>
</tbody>
</table>
A module is a container. It is the basic building block of SystemC

- Similar to VHDL “entity”

Module interface in the header file (ending .h)

Module functionality in the implementation file (ending .cpp)

Module contains:
- Ports
- Internal signal variables
- Internal data variables
- Processes of different types
- Other methods
- Instances of other modules
- Constructor
Module Syntax

Syntax:
SC_MODULE(module_name){
    // body of module
};

EXAMPLE:
SC_MODULE(my_module) {
    // body of module
};

Note: SC_MODULE is a macro for
struct module_name : sc_module

Note the semicolon at the end of the module definition

Code in header file: my_module.h
Basic Modeling Structure

Signal connecting modules (declared in module)

Ports bound directly to each other (no signal)

Module Instance

Module Instance

Ports

Process

Other Method

Other methods can read/write ports and signals

Process can read/write ports and signals

SC_MODULE
{
    Ports
    Signals
    Variables
    Constructor
    Processes
    Modules
};
Ports are the external interface of a module
- Pass information to and from a module
- There are three different kinds of ports
  - input
  - output
  - inout
- Ports are always bound to signals
  - exception: port-to-port binding (explained later)
- Ports are members of the module
- Each port has a data type
  - passed as template parameter
- Ports have to be declared inside a module
  - The direction of the port is specified by the port type
    - input: `sc_in<>`
    - output: `sc_out<>`
    - inout: `sc_inout<>`
  - The data type is passed as template parameter

Declaration as data members of the module:
```c
SC_MODULE (my_mod) {
    sc_in<t_data> port_name;
    sc_out<t_data> port_name;
    sc_inout<t_data> port_name;
}
```

```c
t_data:
    data type of the port
```
Signals - Overview

- Signals are used for communication
  - Exchange of data in a concurrent execution semantics
    - between modules
    - between processes
  - There is only one kind of signal
  - Signals always carry a value
  - Signals need not necessarily be bound to ports
    - may also be used for communication between processes
  - Signals may be
    - members of a module (used for internal communication)
    - used at top-level to connect the modules
- Each signal has a data type
  - passed as template parameter
Signals - Declaration

- Signals are declared inside a module or at top level
  - There is only one type of signal
    - `sc_signal<>`
  - The data type is passed as template parameter

Declaration as data members of the module:
```
SC_MODULE (my_mod) {
    sc_signal<t_data>    signal_name;
};
```

- `t_data:`
  - data type of the signal
Declaring ports and signals within a module
- ports and signals are data members of the module
- good style to declare
  - ports at the beginning of the module
  - signals after port declaration

```sc_module
SC_MODULE(module_name) {
    // ports
    sc_in<int> a;
    sc_out<bool> b;
    sc_inout<sc_bit> c;

    // signals
    sc_signal<int> d;
    sc_signal<char> e;
    sc_signal<sc_int<10> > f;

    // rest
};
```

Note: a space is required by the C++ compiler
To read a value from a port or signal
- use assignment
- use the `.read()` method (recommended)

To write a value to a port or signal
- use assignment
- use the `.write()` method (recommended)

The usage of `.read()` and `.write()` avoids implicit type conversion, which is a source of many compile time errors.

```cpp
sc_signal<int> sig;
int a;
...
a = sig;
sig = 10;
```

```cpp
sc_signal<int> sig;
int a;
...
a = sig.read();
sig.write(10);
```
Data Variables

- Data variables are member variables of a module
  - any legal C/C++ and SystemC/user defined data type
  - internal to the module
  - should not be used outside of the module
  - useful to store the internal state of a module
    - e.g. usage as memory

Example:

```c++
SC_MODULE(count) {
    // ports & signals not shown
    int count_val; // internal data storage
    sc_int<8> mem[512] // array of sc_int
    // Body of module not shown
};
```
Each module has a constructor
- Called at the instantiation of the module
  - initialize internal data structure (e.g. check port-signal binding)
  - initialize all data members of the module to a known state
- Instance name passed as argument
  - useful for debugging/error reporting
- Processes are registered inside the constructor (more later)
- Sub-modules are instanciated inside the constructor (more later)
Module Constructor - Example

Example:

SC_MODULE (my_module) {
    // Ports, internal signals, processes, other methods
    // Constructor
    SC_CTOR(my_module) {
        // process registration & declarations of sensitivity lists
        // module instantiations & port connection declarations
    }
};

Note: SC_CTOR is a macro for
typedef name SC_CURRENT_USER_MODULE; \\name(sc_module_name)
## Agenda

<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>2.1</td>
<td>Data Types</td>
</tr>
<tr>
<td>2.1</td>
<td>Modules, Ports and Signals</td>
</tr>
<tr>
<td>2.2</td>
<td>Processes</td>
</tr>
<tr>
<td>2.3</td>
<td>Hierarchy</td>
</tr>
</tbody>
</table>
Process Properties - 1

- Functionality of a SystemC model is described in processes.

- Processes are member functions of a module. They are registered to the SystemC simulation kernel.
  - Processes are called by the simulation kernel, whenever a signal in their sensitivity list is changing.
  - Some processes execute when called and return control to the calling function (behave like a function).
  - Some processes are called once, and then can suspend themselves and resume execution later (behave like threads).

- Processes are not hierarchical
  - Cannot have a process inside another process (use module for hierarchical design)
Process Properties - 2

- Processes use signals to communicate with each other.
  - Don’t use global variables - may cause order dependencies in code (very bad).

- Processes use timing control statements to implement synchronization and writing of signals in processes (explained later).

- Process and signal updates follow the evaluate-update semantics of SystemC.

- Processes are registered to the simulation kernel within the module’s constructor.
SystemC Processes

- **SC_METHOD** (sc_async_fprocess)
  - process executes everything and wakes up at next event
  - good for modeling combinational logic
  - synchronous logic is sensitive to clock edge

- **SC_THREAD** (sc_async_tprocess)
  - execution is suspended on wait() until next event

- **SC_CTHREAD** (sc_sync_tprocess)
  - execution is suspended on `wait()` until next active clock edge
  - within one process can be only registers sensitive to one clock edge
Async. Function Process Characteristics

- Asynchronous Function Process **SC_METHOD**:  
  - Is sensitive to a set of signals  
    - This set is called *sensitivity list*  
    - May be sensitive to any change on a signal  
    - May be sensitive to the positive or negative edge of a boolean signal  
  - Is invoked whenever any of the inputs it is sensitive to changes.  
  - Once an *asynchronous function process* is invoked:  
    - Entire body of the block is executed.  
    - Instructions are executed infinitely fast (in terms of internal simulation time).  
    - Instructions are executed in order.

<table>
<thead>
<tr>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅ SC_METHOD</td>
</tr>
<tr>
<td>☐ SC_THREAD</td>
</tr>
<tr>
<td>☐ SC_CTHREAD</td>
</tr>
</tbody>
</table>
Asynchronous Function Process Usage

- **Asynchronous Function Process** `SC_METHOD`:
  - Model combinational logic
    - multiplexing, bit extraction, bit manipulation, arith. operations...
  - Model sequential logic
    - use a `SC_METHOD` process sensitive to only on clock edge to update the registers
    - use another `SC_METHOD` process sensitive to all values to be read within this process to calculate new values
  - Monitor signals as part of a testbench
  - Use for RT level modeling
  - Model event driven systems (architectural level)
Defining the Sensitivity List of a Process

- To define the sensitivity list for any process type use
  - `sensitive` with the () operator
    - takes a single port or signal as an argument
    - e.g. `sensitive(sig1); sensitive(sig2); sensitive(sig3);`
  
  - `sensitive` with stream notation (operator `<<`)
    - takes an arbitrary number of arguments
    - e.g. `sensitive << sig1 << sig2 << sig3;`

- `sensitive_pos` with either () or `<<` operator
  - defines sensitivity to positive edge of boolean signal or clock
  - e.g. `sensitive_pos << clk;`

- `sensitive_neg` with either () or `<<` operator
  - defines sensitivity to negative edge of boolean signal or clock
  - e.g. `sensitive_neg << clk;`
SC_MODULE( my_module ) {
    sc_in_clk clk;
    sc_in<bool> reset_n;

    sc_in<int> in1;
    sc_in<int> in2;

    sc_out<int> out1;
    sc_out<int> out2;

    sc_signal<int> sig1;
    sc_signal<int> sig2;

    int _internal;

    void proc1();

    SC_CTOR( my_module ) {
        SC_METHOD( proc1 );
        sensitive << in1 << sig1;
    }
};

void my_module::proc1() {
    int inc = 5;
    inc++;
    sig2 = in1.read() + sig1 + inc;
}

State of internal variables not preserved => expression evaluates to
sig2 = in1+sig1+6
each time the process is executed.
Async. Thread Process Characteristics

- Asynchronous Thread Process **SC_THREAD**: 
  - Is sensitive to a set of signals
  - This set is called *sensitivity list*
  - May be sensitive to any change on a signal
  - May be sensitive to the positive or negative edge of a boolean signal

- Is reactivated whenever any of the inputs it is sensitive to changes.

- Once an *asynchronous thread process* is reactivated:
  - Instructions are executed infinitely fast (in terms of internal simulation time) until the next occurrence of a `wait()` statement.
  - Instructions are executed in order.
  - The next time the process is reactivated execution will continue after the `wait()` statement.
Asynchronous Thread Process **SC_THREAD**:
- May be used to model synchronous AND asynchronous behavior (even in the same process).
- Most useful at higher levels of abstraction (esp. with SystemC 2.0)
- Has to be replaced by asynchronous function or synchronous thread process during refinement.
- May be used in test benches.
Timing Control Statements

- Implement synchronization and writing of signals in processes.
  - `wait()` suspends execution of the process until the process is invoked again.

- If *no* timing control statement used:
  - Process executes in zero time.
  - Outputs are never visible.

- To write to an output signal, a process needs to invoke a timing control statement.

- Multiple interacting synchronous processes must have at least one timing control statement in every path.
Thread Processes: wait() Function

- `wait()` may be used in both `SC_THREAD` and `SC_CTHREAD` processes but NOT in an `SC_METHOD` process (block).

- `wait()` suspends execution of the process until the process is invoked again.

- `wait(<pos_int>)` may be used to wait for a certain number of cycles (`SC_CTHREAD` only).

- In Synchronous process (`SC_CTHREAD`)
  - Statements before the `wait()` are executed in one cycle.
  - Statements after the `wait()` are executed the next cycle.

- In Asynchronous process (`SC_THREAD`)
  - Statements before the `wait()` are executed in last event.
  - Statements after the `wait()` are executed the next event.

Examples:

```
wait();  // waits 1 cycle in synchronous process
wait(4);  // waits 4 cycles in synchronous process
wait(4);  // ERROR!! in Asynchronous process
```
SC_MODULE( my_module ) {
    sc_in_clk clk;
    sc_in<bool> reset_n;

    sc_in<int> in1;
    sc_in<int> in2;

    sc_out<int> out1;
    sc_out<int> out2;

    sc_signal<int> sig1;
    sc_signal<int> sig2;

    int _internal;

    void proc2();

    SC_CTOR( my_module ) {
        SC_THREAD( proc2 );
        sensitive << in2 << sig2;
    }
};

void my_module::proc2()
{
    // start-up functionality goes here
    int inc = 1;

    while(1) {
        out1.write(in2.read() + sig2 + inc);
        ++inc;
        wait();
    }
}

State of internal variables is preserved =>
the value of inc is incremented by one
each time the process is reactivated.
Sync. Thread Processes Characteristics - 1

- **Synchronous process** `SC_CTHREAD`:
  - Sensitive only to one edge of one and only one clock
  - Called the *active edge*

- Special case of a `SC_THREAD` process
  - Cannot use multiple clocks with a synchronous process

- Triggered only at active edge of its clock
  - Therefore inputs are sampled only at the active edge.

- It is not triggered if inputs other than the clock change.

- Models the behavior of unregistered inputs and registered outputs
Synchronous process **SC_CTHREAD**:  
Once invoked the statements execute until a `wait()` statement is encountered.  
At the `wait()` statement, the process execution is suspended.  
At the next execution, process execution starts from the statement after the `wait()` statement.  
Local variables defined in the process function are saved each time the process is suspended.

**Tip**: Use `wait(<pos_int>);` to wait for multiple clock cycles.
Synchronous Thread Process: I/O Timing

```
while (true) {
    out = in1 + in2;
    wait();
}
```

Processes

- **SC_METHOD**
- **SC_THREAD**
- **SC_CTHREAD**

**SystemC 1.0.x**

<table>
<thead>
<tr>
<th></th>
<th>clock</th>
<th>in1</th>
<th>35</th>
<th>in2</th>
<th>13</th>
<th>18</th>
<th>out</th>
<th>14</th>
</tr>
</thead>
</table>

**SystemC 2.0**

<table>
<thead>
<tr>
<th></th>
<th>clock</th>
<th>in1</th>
<th>35</th>
<th>in2</th>
<th>13</th>
<th>18</th>
<th>out</th>
<th>14</th>
<th>53</th>
</tr>
</thead>
</table>

© FhG IIS-A, 2001
Synchronous Thread Process Usage

- **Synchronous Thread Process** `SC_CTHREAD`:
  - Useful for high level synthesis.
  - Modeling of synchronous systems.
  - Modeling of sequential logic.
  - Models the behavior of a block with unregistered inputs and registered outputs.
  - Useful for modeling drivers in test benches.
SC_MODULE( my_module ) {
    sc_in_clk clk;
    sc_in<bool> reset_n;

c_sc_in<int> in1;
    sc_in<int> in2;

c_sc_out<int> out1;
    sc_out<int> out2;

c_sc_signal<int> sig1;
    sc_signal<int> sig2;

    int _internal;

    void proc3();

    SC_CTOR( my_module ) {
        SC_CTHREAD( proc3, clk.pos() );
    }
};

void my_module::proc3()
{
    // start-up functionality goes here
    int inc = 1;

    while(1) {
        out2.write(inc);
        sig2 = sig2 + inc;
        ++inc;
        wait();
    }
}

State of internal variables is preserved =>
the value of inc is incremented by at each
positive clock edge.
<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>2.1</td>
<td>Data Types</td>
</tr>
<tr>
<td>2.1</td>
<td>Modules, Ports and Signals</td>
</tr>
<tr>
<td>2.2</td>
<td>Processes</td>
</tr>
<tr>
<td>2.3</td>
<td>Hierarchy</td>
</tr>
</tbody>
</table>
Systems are modeled hierarchically
- SystemC has to provide a mechanism for hierarchy
- Solution
- Modules may contain instances of other modules
Modeling Hierarchy - Define Internal Signals

SC_MODULE(top)
{
    sc_in<int> in;
    sc_out<int> out;

    sc_signal<int> sig;

    sub_a* inst1;
    sub_b* inst2;
    ....
};

- Define member variables for all the internal signals that are needed to connect the child module’s ports to each other.
- Note: a port of the parent module is directly bound to a port of the child module (no signal needed).
Define member variables that are pointers to the child module’s class for all instances
Create the child module instances within the parent module’s constructor using the C++ keyword `new`.

An arbitrary name is given to the module instances. It is good practice to use the name of the instance variable.
SC_CTOR {
    inst1 = new sub_a("inst_1");
    inst2 = new sub_b("inst_2");
    (*inst1).a_out(sig);
    (*inst1).a_in(in);
    (*inst2)(sig, b_out);
}

- Connect ports of child modules to one another with internal signals.
- Connect ports of the parent module to ports of the child module using direct port-to-port connection. A signal MUST NOT be used.
- Mapping by name is recommended.
- Mapping by position may also be used.
<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why C-based Design Flow</td>
</tr>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>3</td>
<td>SystemC and Synthesis</td>
</tr>
<tr>
<td>4</td>
<td>Simple Example</td>
</tr>
<tr>
<td>5</td>
<td>Resources</td>
</tr>
</tbody>
</table>
SystemC Design Flow

Software

UTF → TF → Partition
abstract RTOS

Hardware

System/Architectural Level
- not synthesizable
- event driven
- abstract communication
- abstract data types

Behavioral Level
- synthesizable
- clocked
- I/O cycle accurate
- algorithmic description

RT Level
- synthesizable
- clocked
- FSM
- data path

© FhG IIS-A, 2001
Available Functionality and Tools

- **System/Architectural Level**
  - channels, interfaces, events, master-slave comm. library (all beta)

- **Behavioral**
  - synthesis to netlist

- **RTL**
  - synthesis to netlist

- **Software**
  - abstract RTOS (not yet available)
Refinement for Synthesis

- **Refine Structure**
  - use signals for I/F
  - partition into synthesizable blocks
  - restrict to synthesizable language subset

- **Refine Control**
  - specify throughput/latency
  - specify I/O protocol

- **Refine Data**
  - restrict to physically meaningful data types (no abstract data)
  - specify bit width
Behavioral Modeling - System

CPU
algorithmic

Memory
algorithmic

Bus Arbiter

I/O Board
algorithmic

Disk
algorithmic

Clk driven Bus IF

Clk driven Bus IF

Clk driven Bus IF

Clk driven Bus IF
Behavioral Modeling - Module

- Interface cycle accurate
  - ports and signals
  - synthesizable data types
  - introduction of a clock

- Rest of module
  - algorithmic
  - clocked (SC_CTHREAD)
  - restriction to synthesizable SystemC subset

- Behavioral Code may utilize
  - functions to reduce code complexity
  - loops
  - conditional statements
  - no abstract data types for ports/signals
  - only synthesizable subset of SystemC

SC_MODULE(fir) {
  sc_in_clk clk;
  sc_in<bool> reset;
  sc_in<T_DATA> in;
  sc_in<bool> in_data_valid;
  sc_out<T_DATA> out;
  sc_out<bool> out_data_valid;
  ...
  SC_CTOR(fir) {
    SC_CTHREAD(fir_process, clk.pos());
    ...
  }
};
RT Level Modeling - System
RT Level Modeling - Module

SC_MODULE(fir) {
    sc_in_clk clk;
    sc_in<bool> reset;
    sc_in<T_DATA> in;
    sc_in<bool> in_data_valid;
    sc_out<T_DATA> out;
    sc_out<bool> out_data_valid;
    ...
    sc_signal<T_STATE> state_curr, state_nxt;
    ...
    SCCTOR(fir) {
        SC_METHOD(fir);
        sensitive << state_curr << in;
        SC_METHOD(fsm);
        sensitive << state_curr << data_valid;
        SC_METHOD(reg);
        sensitive_pos << clk;
        sensitive << reset;
        ...
    }
};

- Interface cycle accurate
  - ports and signals
  - synthesizable data types
  - introduction of a clock

- Rest of module
  - cycle accurate
  - sequential and combinational logic (SC_METHOD)
  - restriction to synthesizable SystemC subset

- RTL Code may utilize
  - functions to reduce code complexity
  - loops
  - conditional statements
  - no abstract data types for ports/signals
  - only synthesizable subset of SystemC
## Agenda

<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why C-based Design Flow</td>
</tr>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>3</td>
<td>SystemC and Synthesis</td>
</tr>
<tr>
<td>4</td>
<td>Simple Example</td>
</tr>
<tr>
<td>5</td>
<td>Resources</td>
</tr>
</tbody>
</table>
Simple Example - Overview
// header file adder.h
typedef int T_ADD;

SC_MODULE(adder) {
    // Input ports
    sc_port<sc_fifo_in_if<T_ADD> > in1;
    sc_port<sc_fifo_in_if<T_ADD> > in2;

    // Output ports
    sc_port<sc_fifo_out_if<T_ADD> > out;

    // Constructor
    SC_CTOR(adder) {
        SC_THREAD(main);
    }

    // Functionality of the process
    void main() {
    }
};
// Implementation file adder.cpp
#include "systemc.h"
#include "adder.h"

void adder::main()
{
    while (true) {
        out->write(in1->read() + in2->read());
        // assign a run-time to process
        wait(10, SC_NS);
    }
}
// header file adder.h

typedef sc_int<8> T_ADD;

SC_MODULE(add) {
    // Clock introduced
    sc_in_clk clk;
    // Input ports
    sc_in<T_ADD> in1;
    sc_in<T_ADD> in2;
    // Output port
    sc_out<T_ADD> out;

    // Constructor
    SC_CTOR(add) {
        SC_CTHREAD(main, clk.pos());
    }

    // Functionality of the process
    void main();
};
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::main()
{
    // initialization
    T_ADD __in1 = 0; T_ADD __in2 = 0;
    T_ADD __out = 0;
    out.write(__out);
    wait();

    // infinite loop
    while(1) {
        __in1 = in1.read();
        __in2 = in2.read();
        __out = __in1 + __in2;
        out.write(__out);
        wait();
    }
}
// header file adder.h

typedef sc_int<8> T_ADD;

SC_MODULE(adder) {
    // Clock introduced
    sc_in_clk clk;
    // Input ports
    sc_in<T_ADD> in1;
    sc_in<T_ADD> in2;
    // Output port
    sc_out<T_ADD> out;
    // internal signal
    sc_signal<T_ADD> sum;
    // Constructor
    SC_CTOR(adder) {
        SC_METHOD(add); sensitive << in1 << in2;
        SC_METHOD(reg); sensitive_pos << clk;
    }
    // Functionality of the process
    void add();
    void reg();
};
// Implementation file adder.cc
#include "systemc.h"
#include "adder.h"

void adder::add()
{
    T_ADD __in1 = in1.read();
    T_ADD __in2 = in2.read();
    sum.write( __in1 + __in2 );
}

void adder::reg()
{
    out.write( sum );
}
## Agenda

<table>
<thead>
<tr>
<th>Unit</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Why C-based Design Flow</td>
</tr>
<tr>
<td>2</td>
<td>Language Elements of SystemC</td>
</tr>
<tr>
<td>3</td>
<td>SystemC and Synthesis</td>
</tr>
<tr>
<td>4</td>
<td>Simple Example</td>
</tr>
<tr>
<td>5</td>
<td>Resources</td>
</tr>
</tbody>
</table>
Welcome to the SystemC Community

Everything you wanted to know about SystemC.

The Open SystemC Initiative is here to establish a modeling platform that enables, promotes and accelerates system-level co-design and IP exchange. Through open community licensing, all companies participate in rapid growth of system-level design.

HOT NEWS

- Design Automation Conference - 2000
- SystemC Around the World NEW!
- Version 1.1 Beta-Download NOW!! NEW!

www.systemc.org
- download
- news
- forum (mailing list)
- FAQs
For SystemC training in Munich, Erlangen, or on-site look at:

- http://www.iis.fhg.de/kursbuch/kurse/systemc.html

Next scheduled training: Nov 21-23 in Munich, Germany

Covers SystemC 2.0