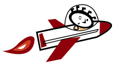
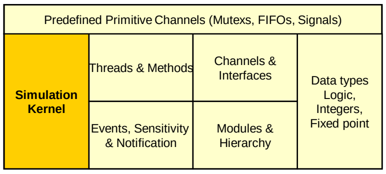
Plan



- Ch1 – Overview of System Design Using SystemC
- Ch2 – Overview of SystemC
- Ch3 – Data Types
- Ch4 – Modules
- Ch5 – Notion of Time
- Ch6 – Concurrency
- Ch7 – Predefined Channels
- Ch8 – Structure
- Ch9 – Communication
- Ch10 – Custom Channels and Data
- Ch11 – Transaction Level Modeling

École polytechnique de l'université de Nice-Sophia Antipolis
Département Electronique



Predefined Primitive Channels (Mutexs, FIFOs, Signals)

Simulation Kernel

Threads & Methods

Channels & Interfaces

Data types Logic, Integers, Fixed point

Events, Sensitivity & Notification

Modules & Hierarchy

Notion of Time

- sc_time Data Type
- Elaboration and Simulation
- wait() Method

sc_time Type

- Goals
  - to measure time
  - to specify a time (waiting, …)
- VHDL : “time” type

Units

SC_SEC	seconds
SC_MS	milliseconds
SC_US	microseconds
SC_NS	nanoseconds
SC_PS	picoseconds
SC_FS	femtoseconds

default : t1(0, SC_SEC)

```
1  sc_time measure, current, last_clk;

2  sc_time period (8.2, SC_NS);  // period = 8.2 ns

3  sc_time clk(period);  // clk = 8.2 ns

   last_clk = sc_time(2, SC_US);  // last_clk = 2 us

   measure = current - last_clk;
   if (measure > hold)
        cerr << “error: setup violation !” << endl;
```

sc_time Class Definition

```
enum sc_time_unit {SC_FS = 0, SC_PS,
                   SC_NS, SC_US, SC_MS, SC_SEC};
class sc_time
{
public:
    sc_time();
    sc_time( double , sc_time_unit );
    sc_time( const sc_time& );

    sc_time& operator= ( const sc_time& );

    sc_dt::uint64 value() const;
    double to_double() const;
    double to_seconds() const;

    const std::string to_string() const;

    bool operator== ( const sc_time& ) const;
    bool operator!= ( const sc_time& ) const;
    bool operator< ( const sc_time& ) const;
    bool operator<= ( const sc_time& ) const;
    bool operator> ( const sc_time& ) const;
    bool operator>= ( const sc_time& ) const;
    sc_time& operator+= ( const sc_time& );
    sc_time& operator-= ( const sc_time& );
    sc_time& operator*= ( double );
    sc_time& operator/= ( double );

    void print( std::ostream& = std::cout ) const;
};
```

constructors

converting functions

operator overloading
(Member Functions)

```
const sc_time operator+ ( const sc_time&, const sc_time& );
const sc_time operator- ( const sc_time&, const sc_time& );
const sc_time operator* ( const sc_time&, double );
const sc_time operator* ( double, const sc_time& );
const sc_time operator/ ( const sc_time&, double );
double operator/ ( const sc_time&, const sc_time& );
std::ostream& operator<< ( std::ostream&, const sc_time& );
```

operator overloading
(Non Member Functions)

```
const sc_time SC_ZERO_TIME;
```

equal to sc_time(0, SC_SEC) (delta delay)

```
void sc_set_time_resolution( double, sc_time_unit );
sc_time sc_get_time_resolution();
```

Example

sc_time_class_definition

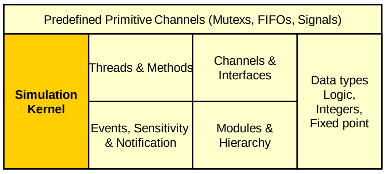
```
#include <systemc.h>

int sc_main(int argc, char* argv[])
{
        sc_time a = sc_time(2.5, SC_US);
        cout << "to_string() : " << a.to_string() << endl;
        cout << "to_double() : " << a.to_double() << endl;
        cout << "to_seconds() : " << a.to_seconds() << endl;
        return 0;
}
```

```
to_string() : 2500 ns
to_double() : 2.5e+006
to_seconds() : 2.5e-006
```

Ecole polytechnique de l'université de Nice-Sophia Antipolis

Département Electronique


Predefined Primitive Channels (Mutexs, FIFOs, Signals)
Simulation Kernel
Threads & Methods
Channels & Interfaces
Data types Logic, Integers, Fixed point
Events, Sensitivity & Notification
Modules & Hierarchy

Notion of Time

- sc_time Data Type
- Elaboration and Simulation
- wait() Method

Methods

- sc_start() method : performs simulation

```
void sc_start();
void sc_start( const sc_time& );
void sc_start( double, sc_time_unit );
```

```
sc_start();                   // Run forever
sc_start(SC_ZERO_TIME);  // Run 1 delta delay
sc_start(8, SC_MS);           // Run 8 ms
```

- sc_stop() method : stop simulation
- sc_time_stamp() method : current time

```
sc_time t = sc_time_stamp() ;
```

- sc_simulation_time() method : current time as a double

```
double t = sc_simulation_time() ;
```

- sc_set_time_resolution() method : resolution (positive power of ten)
- sc_get_time_resolution() method : get the time resolution
- sc_set_default_time_unit() method : default time unit (power of ten)
- sc_get_default_time_unit() method : get default time unit
- sc_delta_count() method : counts the number of delta cycles (return a value of uint64 type)

Methods - Example

startsimulation

```
int sc_main(int argc, char* argv[])
{
      sc_set_time_resolution(1, SC_MS);                    // specified once
      sc_set_default_time_unit(1, SC_SEC);                 // specified once
      cout << "Time resolution is " << sc_get_time_resolution() << endl;       } 1
      cout << "Default time unit is " << sc_get_default_time_unit() << endl;   }

      cout << "current time : " << sc_time_stamp() << endl;     } 2
      cout << "delta number : " << sc_delta_count() << endl;    }
      cout << "sc_start() ..." << endl; <-- 3

      sc_start(7200, SC_SEC);              // 2 Hours

      cout << "current time : " << sc_time_stamp() << endl;     } 4
      cout << "delta number : " << sc_delta_count() << endl;    }

      double t = sc_simulation_time();
      cout << "sc_simulation_time : " << t << endl; <-- 5
      unsigned hours   = int(t / 3600.0);
      t -= 3600.0*hours;
      unsigned minutes = int(t / 60.0);
      t -= 60.0*minutes;
      double   seconds = t;
      cout << hours <<" hours " << minutes << " minutes " ;    } 6
      cout << seconds << " seconds" << endl;                   }

      sc_stop(); <-- 7
      return 0;
}
```

```
Time resolution is 1 ms     } 1
Default time unit is 1 s    }
current time : 0 s          } 2
delta number : 0            }
st_start() ...                  <-- 3
current time : 7200 s       } 4
delta number : 3            }
sc_simulation_time : 7200       <-- 5
2 hours 0 minutes 0 seconds     <-- 6
SystemC: simulation stopped by user.
                                <-- 7
```

Class Hierarchies
C++ Class Inheritance

C++

RECALL … RECALL … RECALL … RECALL

- Classes can inherit members from others classes
  - share a common interface
  - share a common data

```
enum colorType {WHITE, RED, YELLOW, GREEN, BLUE };

class Color
{

protected:
      colorType color;

public:
      Color(colorType c) : color(c)
      { }

      colorType getColor(void) const { return color; }

      void print()
      {
            cout << "Color : " << color << endl;
      }
};
```

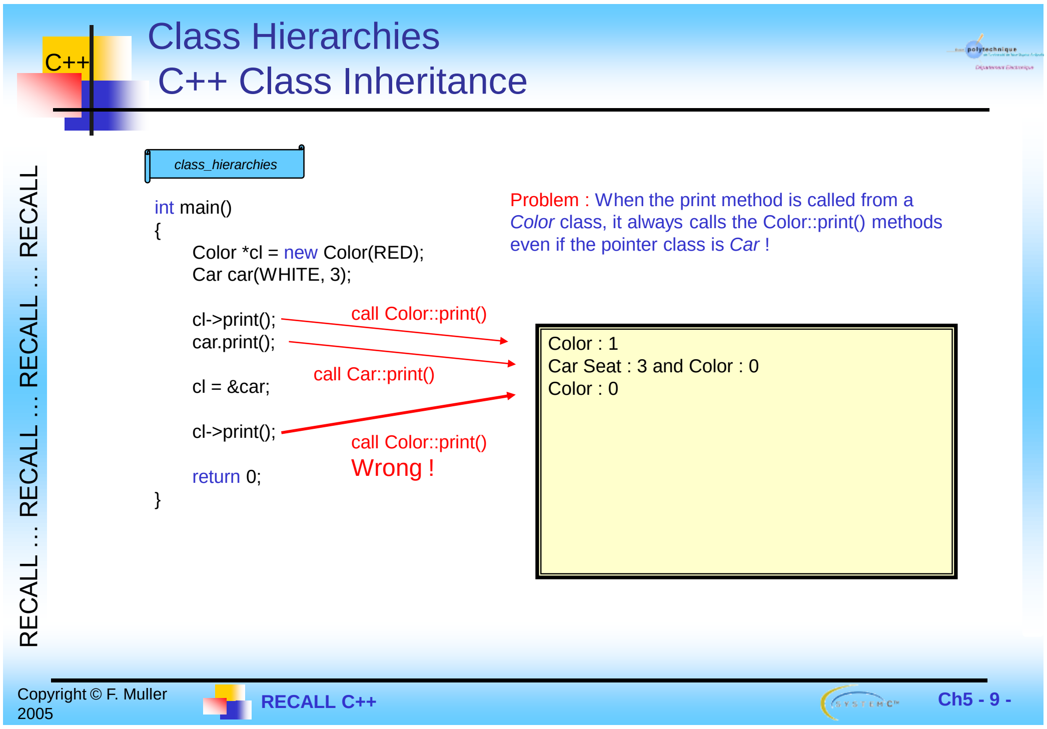
```
class Car : public Color
{

private:
      int seat;

public:
      Car(colorType c, int s) : Color(c), seat(s)
      { }

      int getSeat(void) const { return seat; }

      void print()
      {
            cout << "Car Seat : " << seat ;
            cout << " and Color : " << color << endl;
      }
};
```



Class Hierarchies
C++ Class Inheritance

class_hierarchies

```
int main()
{
    Color *cl = new Color(RED);
    Car car(WHITE, 3);

    cl->print();
    car.print();

    cl = &car;

    cl->print();

    return 0;
}
```

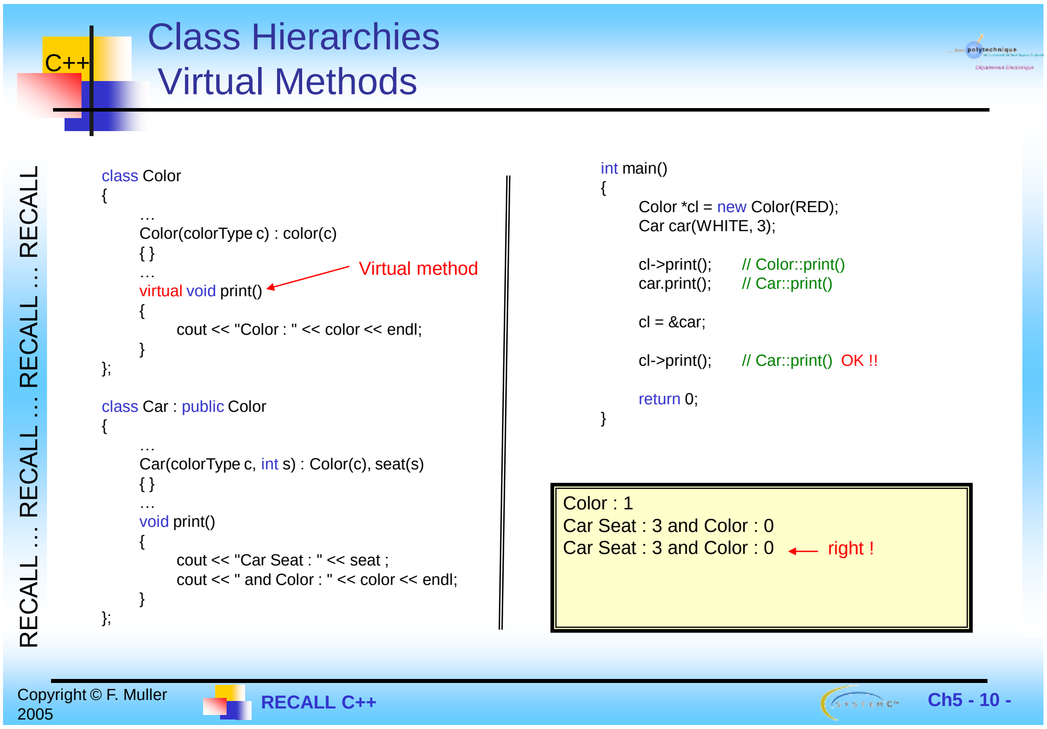
call Color::print()

call Car::print()

call Color::print()
Wrong !

Problem : When the print method is called from a *Color* class, it always calls the Color::print() methods even if the pointer class is *Car* !

```
Color : 1
Car Seat : 3 and Color : 0
Color : 0
```



Class Hierarchies
Virtual Methods

```
class Color
{
    …
    Color(colorType c) : color(c)
    { }
    …
    virtual void print()
    {
        cout << "Color : " << color << endl;
    }
};

class Car : public Color
{
    …
    Car(colorType c, int s) : Color(c), seat(s)
    { }
    …
    void print()
    {
        cout << "Car Seat : " << seat ;
        cout << " and Color : " << color << endl;
    }
};
```

Virtual method

```
int main()
{
    Color *cl = new Color(RED);
    Car car(WHITE, 3);

    cl->print();      // Color::print()
    car.print();      // Car::print()

    cl = &car;

    cl->print();      // Car::print()  OK !!

    return 0;
}
```

```
Color : 1
Car Seat : 3 and Color : 0
Car Seat : 3 and Color : 0
```

right !

C++

Class Hierarchies
C++ Abstract classes

RECALL … RECALL … RECALL … RECALL

- A pure virtual method makes a class abstract
  - a virtual method has no implementation

interface
of print()

```
class Color
{
        …
        virtual void print() = 0 ;  // pure virtual
};
```

- Abstract classes can only be used as base classes

```
Color c(RED);           // ERROR !!
```

- Pure virtual functions are inherited as pure virtual!

implementation
of print()

```
class Car : public Color
{
        …
        void print()
        {
                cout << “Car : “ << endl
                …
};
```

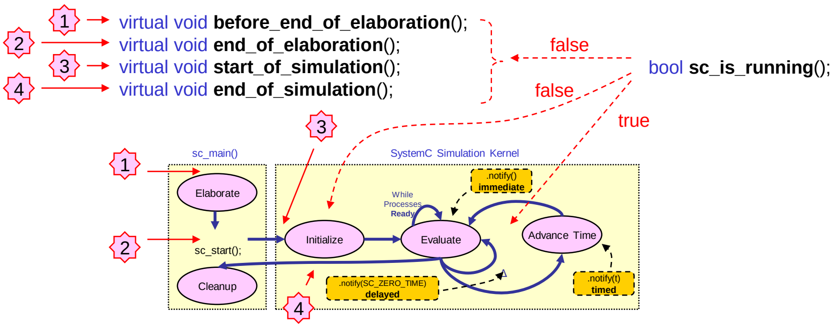
```
int main()
{
        Car car(WHITE, 3);              // OK !
        …
```

Copyright © F. Muller
2005

RECALL C++

Elaboration and Simulation Call Back

- called by the kernel at various stages
  - elaboration
  - simulation
- member functions of class
  - sc_module
  - sc_port, sc_export (Ch10 – Communication)
  - sc_prim_channel (Ch7 – Basic Channels)



```
1 → virtual void before_end_of_elaboration();
2 → virtual void end_of_elaboration();
3 → virtual void start_of_simulation();
4 → virtual void end_of_simulation();
```

bool sc_is_running();

Copyright © F. Muller
2005-2010

Notion of Time

Elaboration and Simulation Call Back Example

eloborate_and_sim

```
SC_MODULE(simple_process)
{
        SC_CTOR(simple_process)
        {
                cout << "   Constructor : " << name() << endl;
                SC_THREAD(my_thread_process);
        }
        void my_thread_process(void) {
process         cout << "   my_thread_process executed within “;
                cout << name() << endl; }

        void before_end_of_elaboration()
        {
                cout << "   before_end_of_elaboration : " << name() << endl;
        }

        void end_of_elaboration() {
                cout << "   end_of_elaboration : " << name() << endl;
        }

        void start_of_simulation() {
                cout << "   start_of_simulation : " << name() << endl;
        }

        void end_of_simulation() {
                cout << "   end_of_simulation : " << name() << endl;
        }
};
```

```
int sc_main(int argc, char* argv[])
{
        cout << "Start main()" << endl;
        simple_process my_instance1("my_inst1");

        cout << "Before start()" << endl;
        sc_start(100, SC_MS);  // Run simulation (100 ms)
        cout << "After start()" << endl;

        sc_stop();
        cout << "After stop()" << endl;
        return 0;
}
```

```
Start main()
   Constructor : my_inst1
Before start()
   before_end_of_elaboration : my_inst1
   end_of_elaboration : my_inst1
   start_of_simulation : my_inst1
   my_thread_process executed within my_inst1
After start()
SystemC: simulation stopped by user.
   end_of_simulation : my_inst1
After stop()
Press any key to continue
```

Ecole polytechnique de l'université de Nice-Sophia Antipolis
Département Electronique

Notion of Time

Predefined Primitive Channels (Mutexs, FIFOs, Signals)			
Simulation Kernel	Threads & Methods	Channels & Interfaces	Data types Logic, Integers, Fixed point
	Events, Sensitivity & Notification	Modules & Hierarchy	

- sc_time Data Type
- Elaboration and Simulation
- wait() Method

wait() Method

- delayed a process for specified periods of time
- used this delay to simulate delays of real activities
  - mechanical actions
  - chemical reaction times
  - signal propagation
- More on wait() (Ch6 – Concurrency)

wait(sc_time t);

wait specified amount of time

wait_method

```
void simple_process::my_thread_process(void)
{
        cout << "Now at " << sc_time_stamp() << endl;
        wait(10, SC_NS);
        cout << "Now at " << sc_time_stamp() << endl;

        sc_time t (5, SC_NS);
        t = t * 3;  // Computes delay
        cout << "delaying " << t << endl;
        wait(t);

        cout << "Now at " << sc_time_stamp() << endl;
}
```

```
Now at 0 ns
Now at 10 ns
delaying 15 ns
Now at 25 ns
```