SISTEME DE CALCUL DEDICATE

Curs 1

Cuprins

- SystemC
  - Istoric
  - ESL
  - Privire de ansamblu
  - Example
- Bibliografie

Istoric

- ***SystemC is a system design language based on C++.***
- SystemC started out as a very restrictive cycle-based simulator and “yet another” RTL language
- the Open SystemC Initiative (OSCI) was formed in 1999

Date	Version	Notes
Sept 1999	0.9	First version; cycle-based
Feb 2000	0.91	Bug fixes
Mar2000	1.0	Widely accessed major release
Oct 2000	1.0.1	Bug fixes
Feb 2001	1.2	Various improvements
Aug 2002	2.0	Add channels & events; cleaner syntax
Apr 2002	2.0.1	Bug fixes; widely used
June 2003	2.0.1	LRM in review
Spring 2004	2.1	LRM submitted for IEEE standard
Dec 2005	2.1v1	IEEE 1666-2005 ratified
July 2006	2.2	Bug fixes to more closely implement IEEE 1666-2005

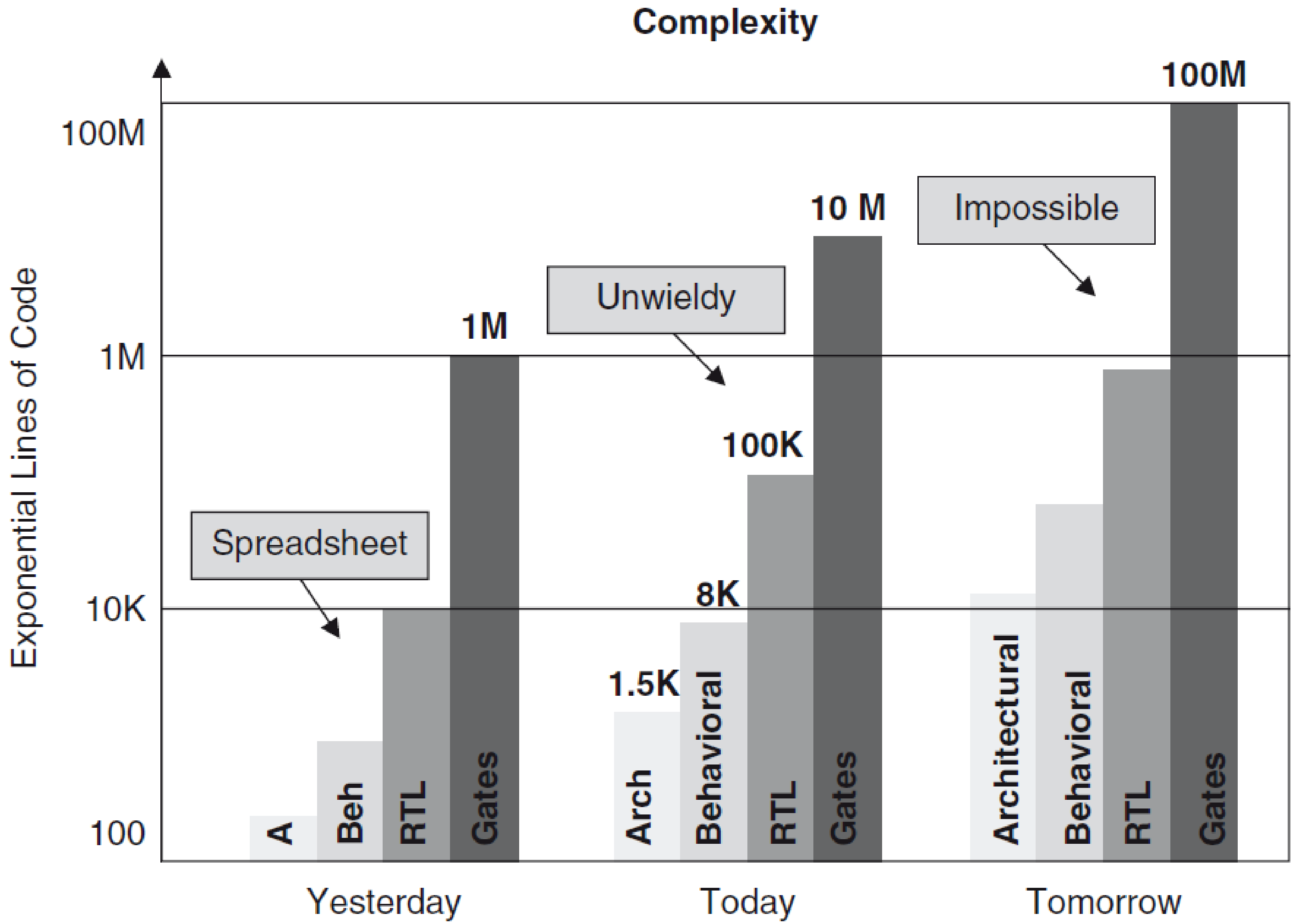
SystemC
electronic system-level design

- SystemC is a system design and modeling language.
- The prevailing name for the concurrent and multidisciplinary approach to the design of complex systems is ***electronic system-level design*** or ESL:
  - ESL happens by modeling systems at higher levels of abstraction
  - Portions of the system model are subsequently iterated and refined, as needed
  - A set of techniques has evolved called ***Transaction-Level Modeling*** or TLM to aide with this task

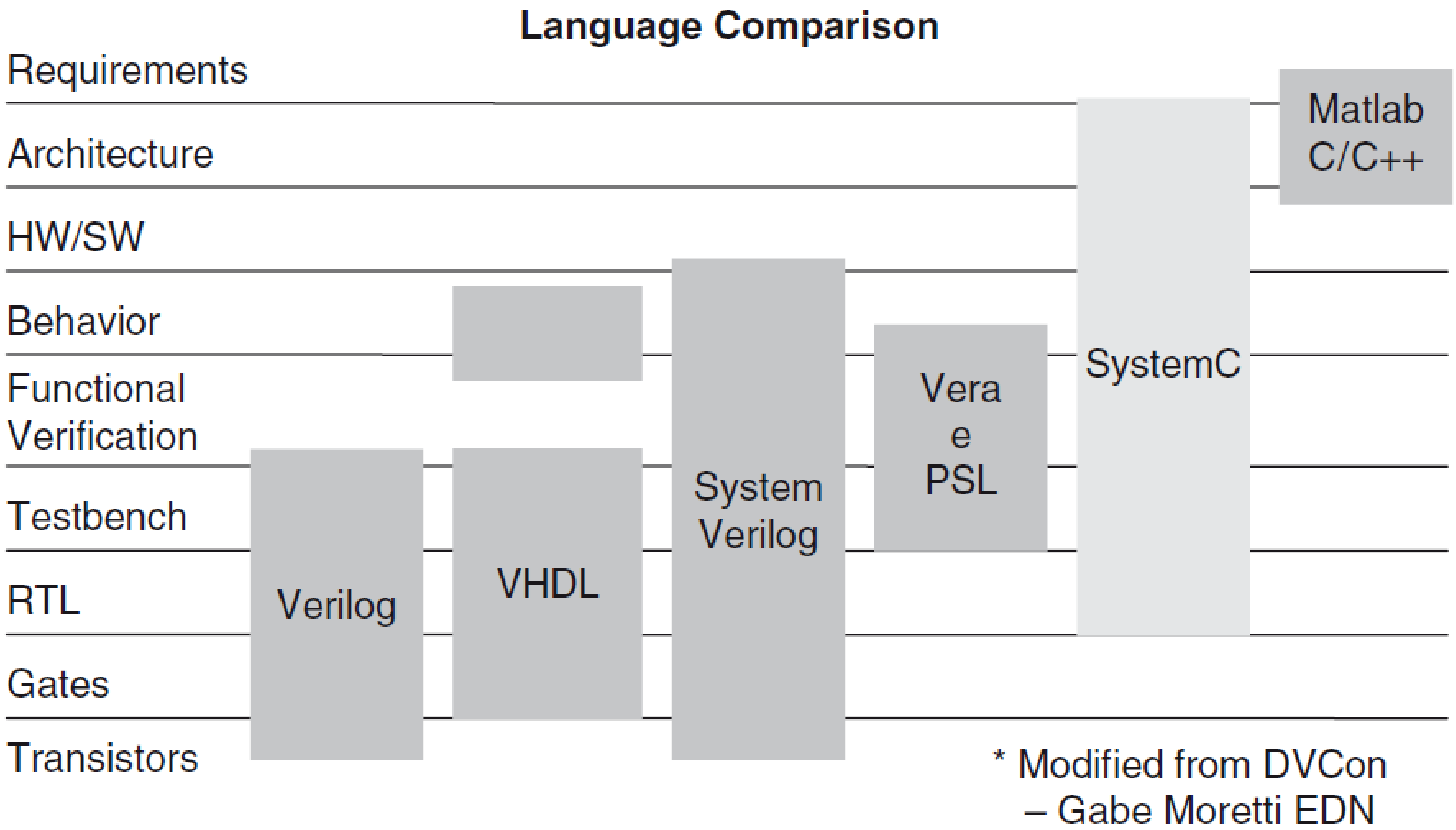
SystemC
electronic system-level design

- ESL techniques evolved in response to increasing design complexity and increasingly shortened design cycles
- System development teams find themselves asking questions like:
  - Should this function be implemented in hardware, software, or with a better algorithm?
  - Does this function use less power in hardware or software?
  - Do we have enough interconnect bandwidth for our algorithm?
  - What is the minimum precision required for our algorithm to work?
- ***Shortened Design Cycle = Need For Concurrent Design***

SystemC
electronic system-level design

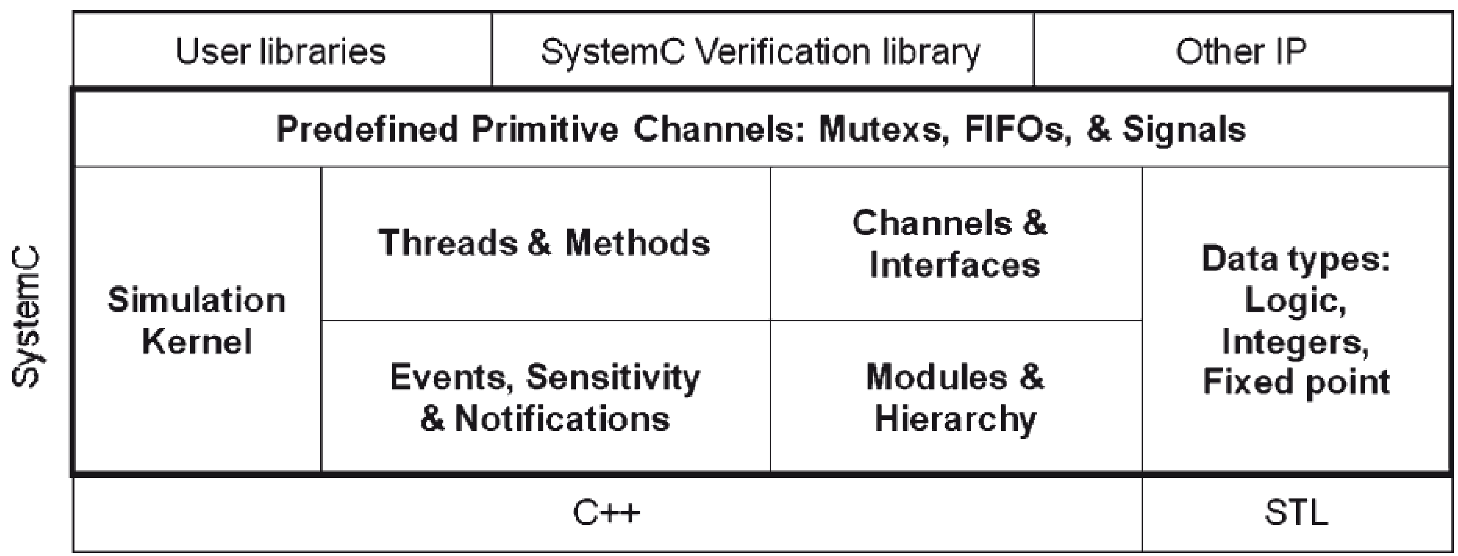


SystemC
electronic system-level design

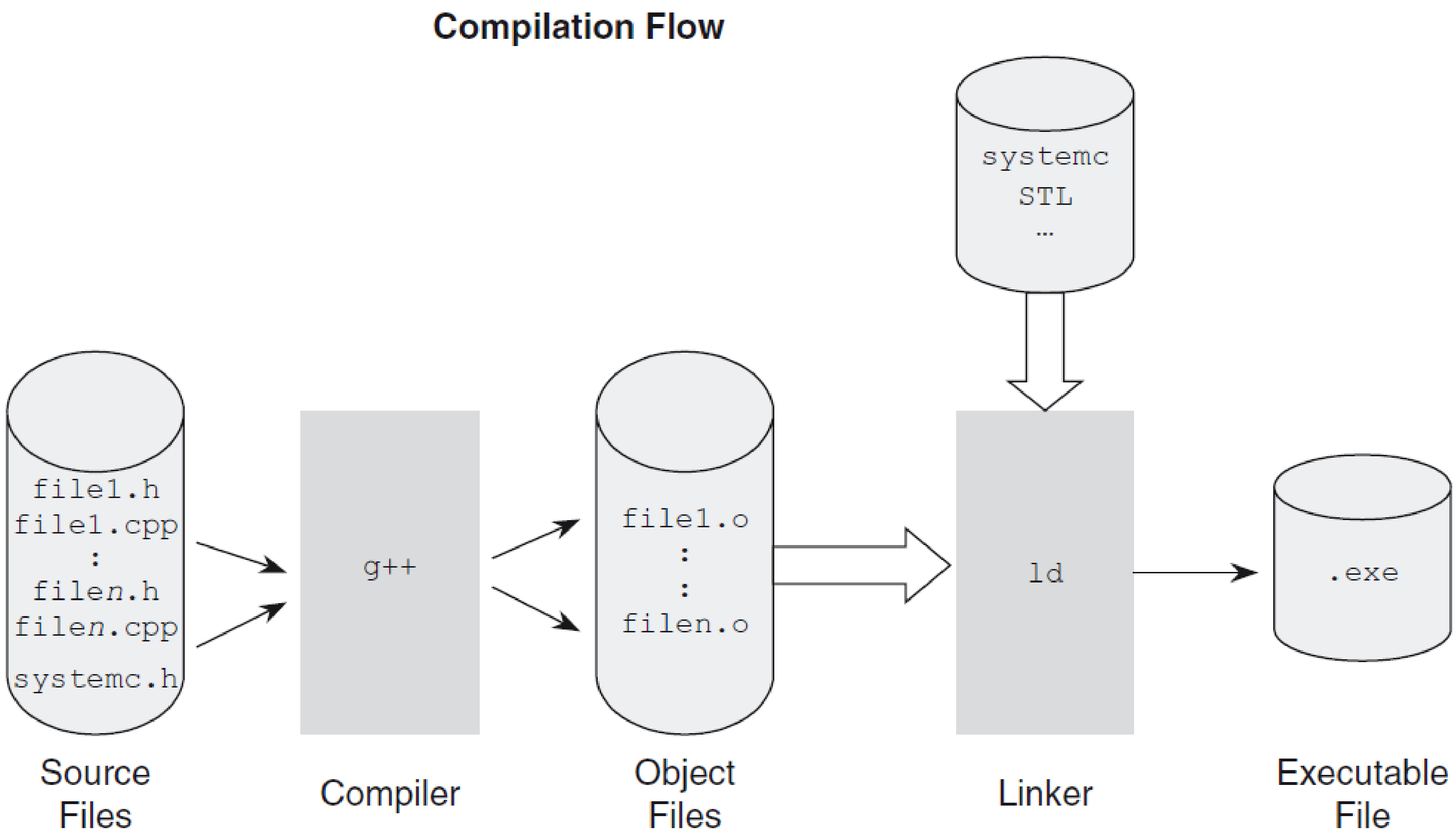


SystemC Overview

- SystemC addresses the modeling of both hardware and software using C++.
- The components of a SystemC environment include a:
  - SystemC-supported platform
  - SystemC-supported C++ compiler
  - SystemC library (downloaded and compiled for your C++ compiler)
  - Compiler command sequence make file or equivalent



SystemC Overview



SystemC Overview

```
#include <systemc>
SC_MODULE(Hello_SystemC) { // declare module class

  SC_CTOR(Hello_SystemC) { // create a constructor
    SC_THREAD(main_thread);// register the process
  }//end constructor

  void main_thread(void) {
    SC_REPORT_INFO(" Hello SystemC World!");
  }
};

int sc_main(int sc_argc, char* sc_argv[]) {

  //create an instance of the SystemC module
  Hello_SystemC HelloWorld_i("HelloWorld_i");

  sc_start(); // invoke the simulator

  return 0;
}
```

SystemC Overview

- The major hardware-oriented features implemented within SystemC include:
  - Time model
  - Hardware data types
  - Module hierarchy to manage structure and connectivity
  - Communications management between concurrent units of execution
  - Concurrency model

SystemC Overview

- **Time Model**
  - SystemC tracks time with 64 bits of resolution using a class known as **sc_time**.
  - Global time is advanced within the kernel.
  - For those models that require a clock, a class called **sc_clock** is provided.
- **Hardware types**
  - SystemC provides hardware-compatible data types that support explicit bit widths for both integral and fixed-point quantities.
  - Digital hardware requires non-binary representation such as tri-state and unknowns, which are provided by SystemC.

SystemC Overview

- **Hierarchy and Structure**
  - For modeling hardware hierarchy, SystemC uses ***the module entity*** interconnected to other modules using channels.
  - The hierarchy comes from the instantiation of module classes within other modules.
- **Communications management**
  - The SystemC channel provides a powerful mechanism for modeling communications.
  - Channels can represent complex communications schemes that eventually map to significant hardware such as the AMBA bus.
  - At the same time, channels may also represent very simple communications such as a wire or a FIFO (first-in first-out queue).

SystemC Overview

- **Concurrency**
  - SystemC provides a simulation kernel
  - Concurrency in a simulator is always an illusion.
  - The simulator uses a cooperative multitasking model.
  - The simulator merely provides a kernel to orchestrate the swapping of the various concurrent elements, called simulation processes.

SystemC Overview

- **Modules and Hierarchy**
  - Hardware designs typically contain hierarchy to reduce complexity.
  - Design components are encapsulated as “modules”.
  - Modules are classes that inherit from the ***sc_module*** base class.
  - Modules may contain other modules, processes, and channels and ports for connectivity.

SystemC Overview

- **SystemC Threads and Methods**
  - Simulation processes are simply member functions of **sc_module** classes that are “registered” with the simulation kernel.
  - They need no arguments and they return no value.
  - From a software perspective, processes are simply threads of execution.
  - From a hardware perspective, processes provide necessary modeling of independently timed circuits.
  - The **SC_METHOD** and **SC_THREAD** are the basic units of concurrent execution.
  - The simulation kernel invokes each of these processes.

SystemC Overview

- **Events, Sensitivity, and Notification**
  - Events, sensitivity, and notification are very important concepts for understanding the implementation of concurrency
  - Events are implemented with the SystemC ***sc_event*** and ***sc_event_queue*** classes
  - SystemC has two types of sensitivity: static and dynamic

SystemC Overview

- **SystemC Data Types**
  - Several hardware data types are provided in SystemC
  - These data types are implemented using templated classes and generous operator overloading
  - Non-binary hardware types are supported with four-state logic (0,1,X,Z) data types (e.g., sc_logic)

SystemC Overview

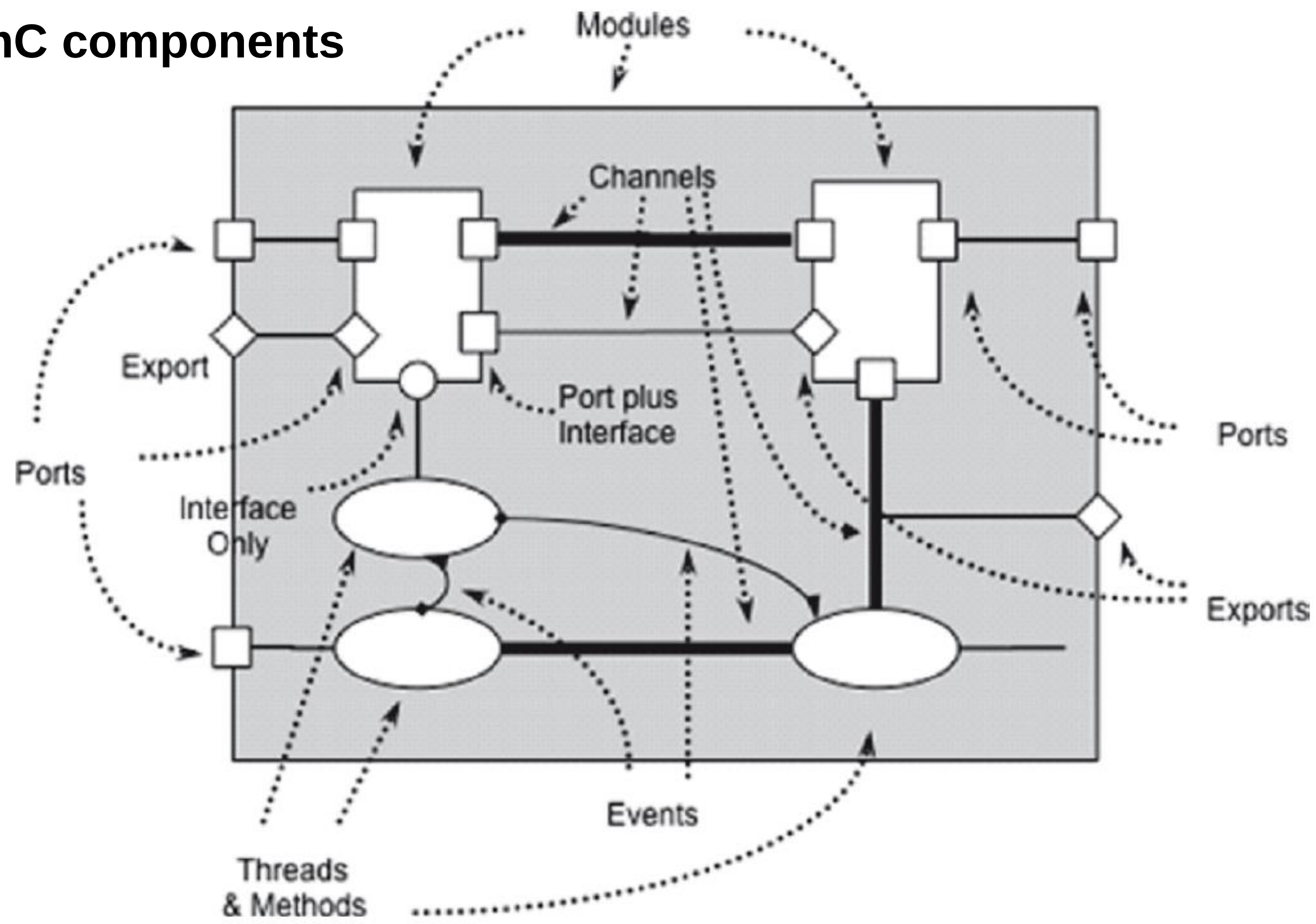
- **SystemC Data Types**
  - Several hardware data types are provided in SystemC
  - These data types are implemented using templated classes and generous operator overloading
  - Non-binary hardware types are supported with four-state logic (0,1,X,Z) data types (e.g., sc_logic)

SystemC Overview

- **Ports, Interfaces, and Channels**
  - Processes need to communicate with other processes both locally and in other modules.
  - In SystemC, processes communicate using channels or events
  - Modules may interconnect using channels, and connect via ports
  - Module interconnection happens programmatically in SystemC during the elaboration phase

SystemC Overview

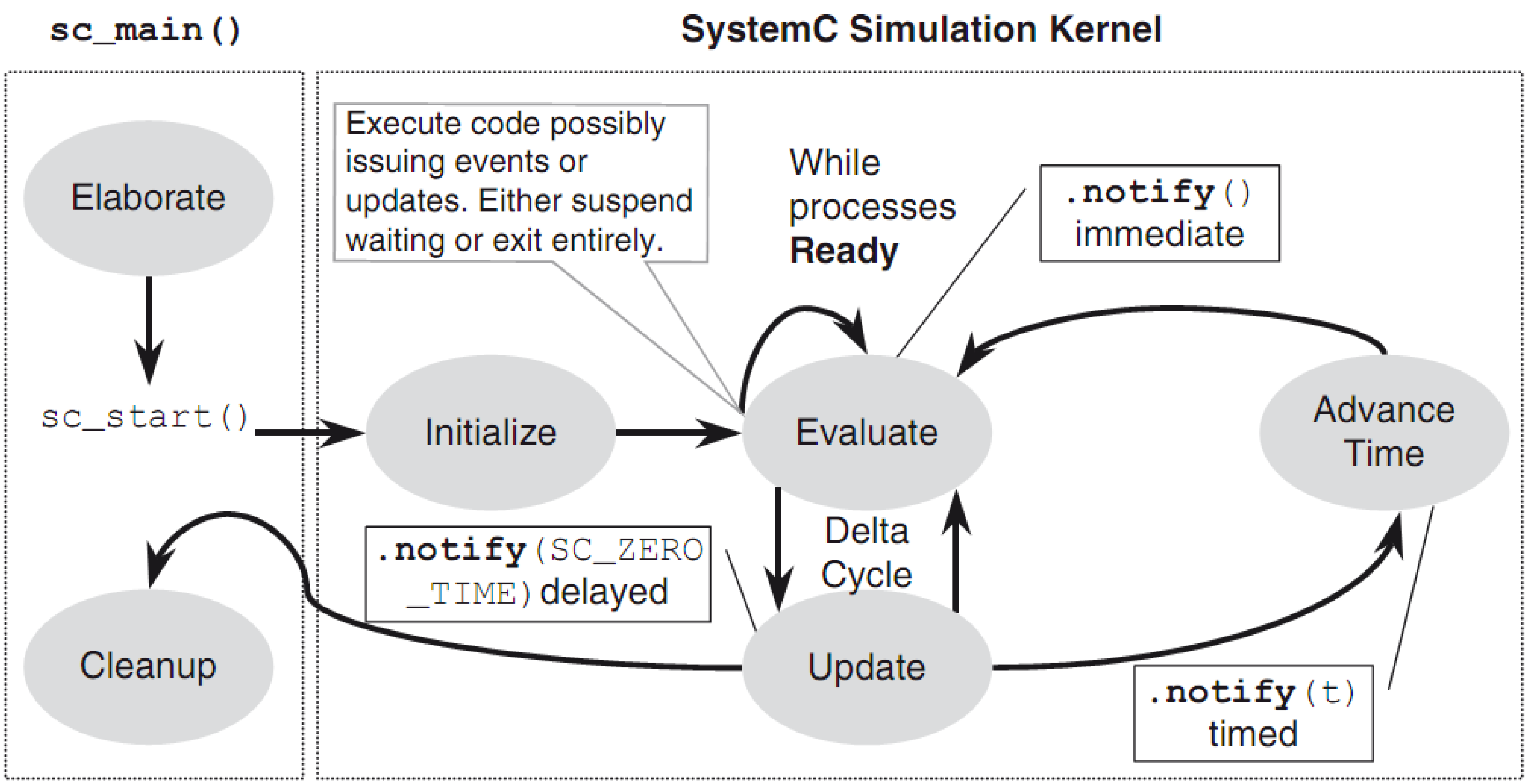
SystemC components



SystemC Overview

- **SystemC Simulation Kernel**
  - The SystemC simulator has two major phases of operation: ***elaboration*** and ***execution***.
  - A third, often minor, phase occurs at the end of execution; this phase could be characterized as ***post-processing*** or ***cleanup***.
  - Execution of statements prior to the ***sc_start()*** function call are known as the elaboration phase.
    - This phase is characterized by the initialization of data structures, the establishment of connectivity, and the preparation for the second phase, execution.
  - The execution phase hands control to the SystemC simulation kernel:
    - orchestrates the execution of processes to create an illusion of concurrency.

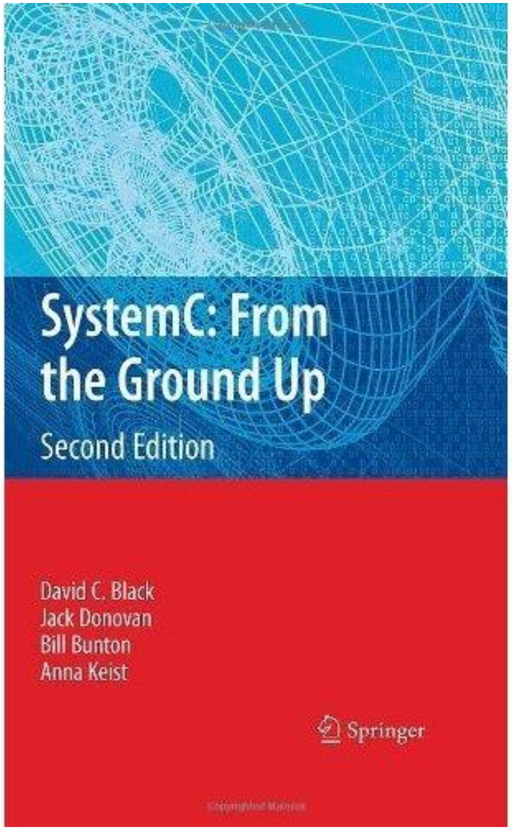
SystemC Overview



Example

- Model and simulate a simple logic gate.
  - Hints (for Visual Studio) – project properties
    - C/C++, General, Additional Include Directories
      - $(SYSTEMC)\..\src
    - C/C++, Code generation, Runtime Library:
      - Multi-threaded Debug (/MTd)
    - C/C++, Language, Enable Run-Time Type Information:
      - Yes (/GR)
    - C/C++, Command Line, Additional Options:
      - /vmg
    - Linker, General, Additional Library Directories
      - $(SYSTEMC)\SystemC\Debug
    - Linker, Input, Additional Dependencies:
      - SystemC.lib

Bibliografie



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”

SISTEME DE CALCUL DEDICATE

Curs 2

Outline

- SystemC
  - Data types
  - Modules
- Bibliografie

Data types

- the use of SystemC data types
  - is not restricted to models using the simulation kernel
- simulation models may be created using any of the available data types
- the choice of data types affects
  - simulation speed
  - synthesizability
  - synthesis results
- the use of the native C++ data types
  - maximize simulation performance
  - decreases hardware fidelity and synthesizability

Data types

- The native C++ data types

Name	Description	Size
`char`	Character	1 byte
`short int (short)`	Short integer	2 bytes
`int`	Integer	4 bytes
`long int (long)`	Long integer	4 bytes
`long long int`	Long long integer	8 bytes
`float`	Floating point	4 bytes
`double`	Double precision floating point	8 bytes

```
// Example native C++ data types
const bool     WARNING_LIGHT(true);// Status
int            spark_offset; // Adjust ignition
unsigned       repairs(0);   // # of repairs
unsigned long  mileage;      // Miles driven
short int      speedometer;  // -20..0..100 MPH
float          temperature;  // Engine temp in C
double         time_of_last_request; // bus activity
string         license_plate;// license plate text
enum           Direction { N,NE,E,SE,S,SW,W,NW };
Direction      compass;
```

Data types

- the SystemC library provides data types for
  - digital logic
  - fixed-point arithmetic
- two SystemC logic vector types
  - 2-valued logic
  - 4-valued logic
- two SystemC numeric types
  - integers
  - fixed-point

Data types

- all of the SystemC data types (except **sc_logic**)
  - length configurable over a range much broader than the native C++ data types
- SystemC provides
  - assignment and initialization operations with type conversions
- SystemC data types
  - implement equality and bitwise operations

Data types

- SystemC arithmetic data types
  - implement arithmetic and relational operations
- SystemC data types
  - allow single-bit and multi-bit select operations
- SystemC data type are part of the **sc_dt** namespace

Data types

- two logic vector types
  - **sc_bv**<*W*> (bit vector)
    - values restricted to logic zero or logic one
  - **sc_lv**<*W*> (logic vector)
    - includes unknown and high impedance (tri-state)
      - logic 0 - **SC_LOGIC_0, Log_0,** or **‘0’**
      - logic 1 - **SC_LOGIC_1, Log_1,** or **‘1’**
      - high-impedance - **SC_LOGIC_Z, Log_Z, ‘Z’** or **‘z’**
      - unknown - **SC_LOGIC_X, Log_X, ‘X’** or **‘x’**
- a single-bit logic type
  - **sc_logic**

Data types

```
sc_bv<5> positions = "01101";
sc_bv<6> mask = "100111";
sc_bv<5> active = positions & mask;// 00101
sc_bv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";// 00001
positions[2] = active[0] ^ flag;
```

```
sc_lv<5> positions =  "01xz1";
sc_lv<6> mask      = "10ZX11";
sc_lv<5> active = positions & mask; // 0xxx1
sc_lv<1> all = active.and_reduce(); // SC_LOGIC_0
positions.range(3,2) = "00";        // 000Z1
positions[2] = active[0] ^ flag;    // !flag
```

Data types

- SystemC integer data types
  - templated
  - may have data widths from 1 to hundreds of bits
  - allow bit selections, bit range selections, and concatenation operations
- **sc_int**<*W*> and **sc_uint**<*W*>
  - provide an efficient way to model data with specific widths from 1- to 64-bits wide
- **sc_bigint**<*W*> and **sc_biguint**<*W*>
  - for modeling larger hardware

Data types

```
// SystemC integer data types
sc_int<5>       seat_position=3; //5 bits: 4 plus
                                 // sign
sc_uint<13>     days_SLOC(4000); //13 bits: no sign
sc_biguint<80>  revs_SLOC;       // 80 bits: no sign
```

Data types

- fixed-point data types
  - address the need for non-integer data types when modeling DSP applications
- multiple fixed-point data types
  - signed and unsigned
  - compile-time (templated) and run-time configurable
  - fixed-precision and limited precision (_fast) versions
- parameters
  - word length (*WL*)
  - integer-word length (*IWL*)
  - quantization mode (*QUANT*)
  - overflow mode (*OVFLW*),
  - and number of saturation bits (*NBITS*)

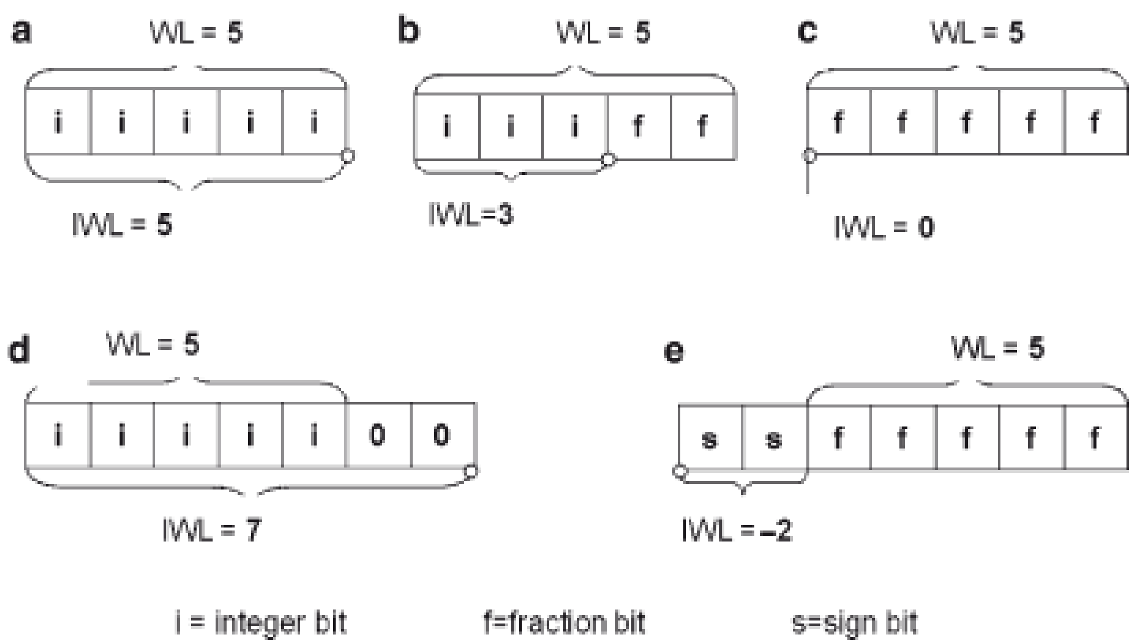
Data types

```
sc_fixed<WL,IWL[,QUANT[,OVFLW[,NBITS]]>      NAME;
sc_ufixed<WL,IWL[,QUANT[,OVFLW[,NBITS]]>     NAME;
sc_fixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]]>  NAME;
sc_ufixed_fast<WL,IWL[,QUANT[,OVFLW[,NBITS]]> NAME;

sc_fix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]]]);
sc_ufix NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]]]);
sc_fix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]]]);
sc_ufix_fast NAME(WL,IWL[,QUANT[,OVFLW[,NBITS]]]);
```

```
// to enable fixed-point data types
#define SC_INCLUDE_FX
#include <systemc>
// fixed-point data types are now enabled
sc_fixed<5,3>    compass // 5-bit fixed-point word
```

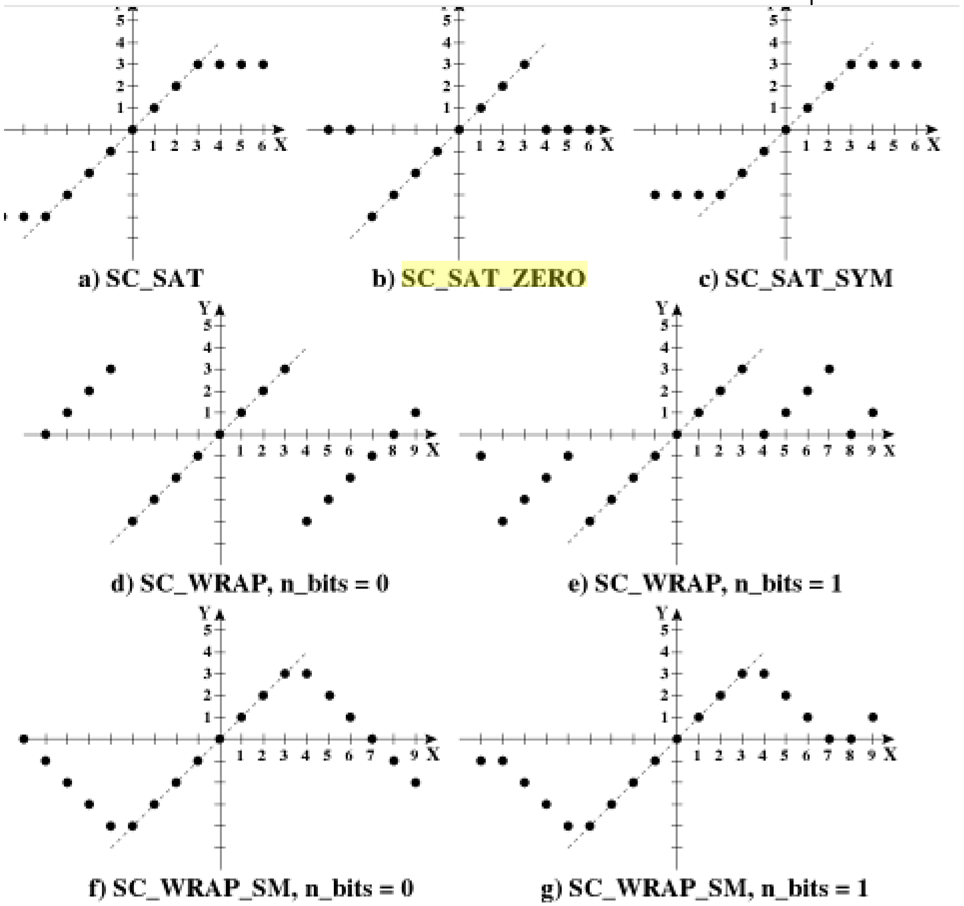
Data types

- example: **sc_fixed**<5,3>
  - represent values from –4.00 up to 3.75 in 1/4 increments



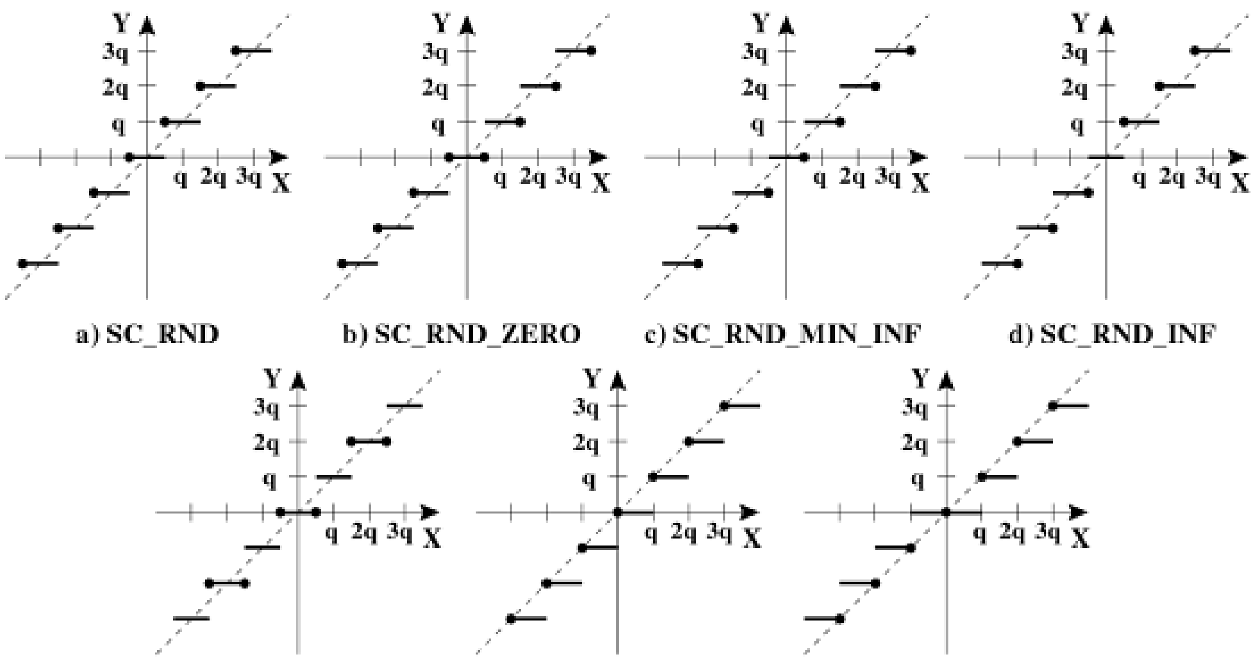
Data types

Name	Overflow Meaning
SC_SAT	Saturate
SC_SAT_ZERO	Saturate to zero
SC_SAT_SYM	Saturate symmetrically
SC_WRAP	Wraparound
SC_WRAP_SYM	Wraparound symmetrically



Data types

Name	Quantization Mode
SC_RND	Round
SC_RND_ZERO	Round towards zero
SC_RND_MIN_INF	Round towards minus infinity
SC_RND_INF	Round towards infinity
SC_RND_CONV	Convergent rounding[a]
SC_TRN	Truncate
SC_TRN_ZERO	Truncate towards zero



a) SC_RND b) SC_RND_ZERO c) SC_RND_MIN_INF d) SC_RND_INF e) SC_RND_CONV f) SC_TRN g) SC_TRN_ZERO

Data types

- SystemC string literals may be used to assign values to any of the SystemC data types
- string literals consist of
  - a prefix
  - a magnitude
  - an optional sign character "+" or "-"
- instances of the SystemC data types
  - may be converted to a standard C++ string

```
string to_string(sc_numrep rep, bool wprefix);
```

Data types

sc_numrep	Prefix	Meaning	sc_int<5> = 13 sc_int<5> = -13
SC_DEC	0d	Decimal	0d13 -0d13
SC_BIN	0b	Binary	0b01101 0b10011
SC_BIN_US	0bus	Binary unsigned	0bus1101 negative
SC_BIN_SM	0bsm	Binary signed magnitude	0bsm01101 -0bsm01101
SC_OCT	0o	Octal	0o15 0o63
SC_OCT_US	0ous	Octal unsigned	0ous15 negative
SC_OCT_SM	0osm	Octal signed magnitude	0osm15 -0osm15
SC_HEX	0x	Hex	0x0d 0xf3
SC_HEX_US	0xus	Hex unsigned	0xusd negative
SC_HEX_SM	0xsm	Hex signed magnitude	0xsm0d -0xsm0d
SC_CSD	0csd	Canonical signed digit	0csd10-01 0csd-010-

Data types

- SystemC data types
  - support all the common operations
    - with operator overloading
- SystemC provides
  - special methods to access bits, bit ranges
  - perform explicit conversions

Data types

Comparison	== != > >= < <=
Arithmetic	++ -- * / % + - << >>
Bitwise	~ & | ^
Assignment	= &= |= ^= *= /= %= += -= <<= >>=

Bit Selection	bit(idx), [idx]
Range Selection	range(high,low), (high,low)
Conversion (to C++ types)	to_double(),to_int(), to_int64(),to_long(), to_uint(),to_uint64(), to_ulong(),to_string(type)
Testing	is_zero(), is_neg(), length()
Bit Reduction	and_reduce(),nand_reduce(), or_reduce(),nor_reduce(), xor_reduce(), xnor_reduce()

Data types

Speed	Data type
Fastest	Native C/C++ Data Types (e.g., int, double and bool)
	sc_int<W>, sc_uint<W>
	sc_bv<W>
	sc_logic, sc_lv<W>
	sc_bigint<W>, sc_biguint<W>
	sc_fixed_fast<WL,IL,…>, sc_fix_fast,
	sc_ufixed_fast<WL,IL,…>, sc_ufix_fast
Slowest	sc_fixed<WL,IL,…>, sc_fix,
	sc_ufixed<WL,IL,…>, sc_ufix

Modules

- all programs need a starting point

```
int main(int argc, char* argv[]) {
    BODY_OF_PROGRAM
  return EXIT_CODE; // Zero indicates success
}
```

```
int sc_main(int argc, char* argv[]) {
  ELABORATION
  sc_start(); // <-- Simulation begins & ends
              //     in this function!
  [POST-PROCESSING]
  return EXIT_CODE; // Zero indicates success
}
```

Modules

- complex systems consist of many independently functioning components
- components may represent
  - hardware
  - software
  - any physical entity

```
#include <systemc>
SC_MODULE(module_name) {
  MODULE_BODY
};
```

Modules

- elements of *MODULE BODY:*
  - Ports
  - Member channel instances
  - Member data instances
  - Member module instances (sub-designs)
  - Constructor
  - Destructor
  - Simulation process member functions (processes)
  - Other methods (i.e., member functions)
- only the constructor is required

Modules

- The **SC_MODULE** constructor performs
  - Initializing/allocating sub-designs
  - Connecting sub-designs
  - Registering processes with the SystemC kernel
  - Providing static sensitivity
  - Miscellaneous user-defined setup
- To simplify coding, SystemC provides the macro, **SC_CTOR**().

Modules

```
SC_MODULE(module_name) {
  SC_CTOR(module_name)
  : Initialization // C++ initialization list
  {
    Subdesign_Allocation
    Subdesign_Connectivity
    Process_Registration
    Miscellaneous_Setup
  }
};
```

Modules

- SystemC simulation process
  - the basic unit of execution
- All simulation processes
  - are registered with the SystemC simulation kernel
  - are called by the kernel, and *only* from the SystemC simulation kernel
- ***DEFINITION***: A SystemC simulation process is a method (member function) of an **SC_MODULE** that is invoked by the scheduler in the SystemC simulation kernel.

```
void PROCESS_NAME(void);
```

Modules

- The most straightforward type of process to understand is the SystemC thread, **SC_THREAD**.
- a simple **SC_THREAD**
  - begins execution when the scheduler calls it
  - may also suspend itself
- a process method
  - must identify and register with the simulation kernel
  - allows the thread to be invoked by the simulation kernel's scheduler
  - the registration occurs within the module class constructor

Modules

```
SC_THREAD(process_name);//Must be INSIDE constructor
```

```
//FILE: basic_process_ex.h
SC_MODULE(basic_process_ex) {
  SC_CTOR(basic_process_ex) {
    SC_THREAD(my_thread_process);
  }
  void my_thread_process(void);
};
```

Modules

- alternative approach to creating constructors
  - uses macro named **SC_HAS_PROCESS**
- use **SC_HAS_PROCESS**
  - constructors with arguments
  - constructor in the implementation

```
//FILE: module_name.h
SC_MODULE(module_name) {
  SC_HAS_PROCESS(module_name);
  module_name(sc_module_name
              instname[, other_args…])
  : sc_module(instname)
  [, other_initializers]
  {
    CONSTRUCTOR_BODY
  }
};
```

Modules

```
#ifndef NAME_H
#define NAME_H
#include "submodule.h"
…
SC_MODULE(NAME) {
    Port declarations
    Channel/submodule instances
    SC_CTOR(NAME)
    : Initializations
     {
        Connectivity
        Process registrations
    }
    Process declarations
    Helper declarations
};
#endif
```

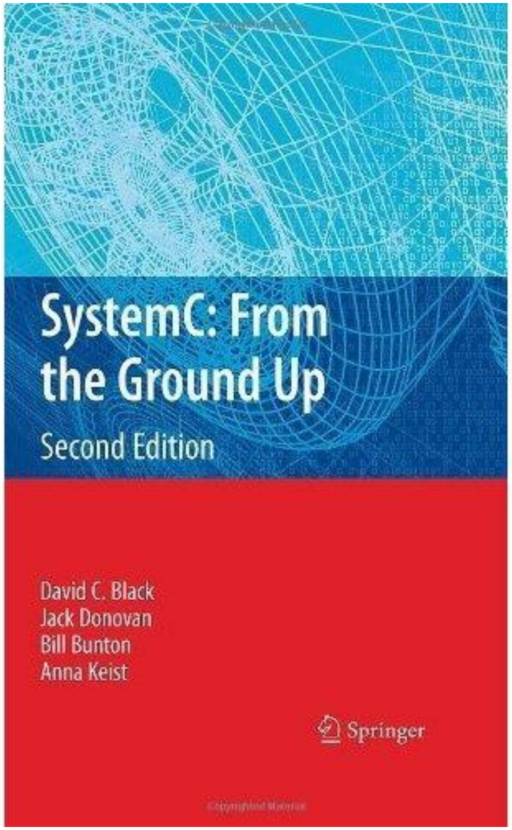
```
#include <systemc>
#include "NAME.h"
NAME::Process {implementations }
NAME::Helper {implementations }
```

Modules

```
#ifndef NAME_H
#define NAME_H
Submodule forward class declarations
SC_MODULE(NAME) {
    Port declarations
    Channel/Submodule* definitions
    // Constructor declaration:
    SC_CTOR(NAME);
    Process declarations
    Helper declarations
};
#endif
```

```
#include <systemc>
#include "NAME.h"
SC_HAS_PROCESS(NAME);
 NAME::NAME(sc_module_name nm)
: sc_module(nm)
, Initializations
{
    Channel allocations
    Submodule allocations
    Connectivity
    Process registrations
}
NAME::Process {implementations }
NAME::Helper {implementations }
```

Bibliografie



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”

SISTEME DE CALCUL DEDICATE

Curs 3

Outline

- SystemC
  - Time
  - Concurrency
- Bibliography

Time

- three unique time measurements:
  - the simulation’s wall-clock time
    - the time from the start of execution to completion, including time waiting on other system activities and applications.
  - the simulation’s processor time
    - the actual time spent executing the simulation, which will always be less than the simulation’s wall-clock time
  - the simulated time
    - the time being modeled by the simulation
      - it may be less than or greater than the simulation’s wall-clock time

Time

- SystemC simulation performance is a combination of many factors:
  - the host system
  - system load
  - the C++ compiler
  - the SystemC simulator
  - the model being

Time

- data type sc_time – used by the simulation kernel
  - to track simulated time
  - to specify delays and timeouts
- sc_time is represented by a minimum of a 64-bit unsigned integer

```
sc_time name…; // no initialization
sc_time name(double, sc_time_unit)…;
sc_time name(const sc_time&)…;
```

Time

- time units
  - are defined by the enumeration **sc_time_unit**

enum	Units	Magnitude
SC_FS	femtoseconds	10^{-15}
SC_PS	picoseconds	10^{-12}
SC_NS	nanoseconds	10^{-9}
SC_US	microseconds	10^{-6}
SC_MS	milliseconds	10^{-3}
SC_SEC	seconds	10^{0}

Time

- all objects of **sc_time** use a single (global) time resolution
  - that has a default of 1 picosecond
  - the **sc_time** class provides get and set methods
  - **sc_set_time_resolution**()
    - may be used to change time resolution once and only once in a simulation
    - the change must occur before both creating objects of **sc_time** and starting the simulation.

```
//positive power of ten for resolution
sc_set_time_resolution(double, sc_time_unit);
```

Time

- objects of sc_time
  - may be used as operands for assignment, arithmetic, and comparison operations
  - provides conversion methods to convert sc_time to a double (to_double()) or to a double scaled to seconds (to_seconds())

```
sc_time t_PERIOD(5, SC_NS);
sc_time t_TIMEOUT(100, SC_MS);
sc_time t_MEASURE, t_CURRENT, t_LAST_CLOCK;
t_MEASURE = (t_CURRENT-t_LAST_CLOCK);
if (t_MEASURE > t_HOLD) { error("Setup violated") }
```

Time

- SystemC simulation kernel tracks simulated time using an sc_time object
  - **sc_time_stamp**() can be used to obtain the current simulated **time_value**

```
sc_time current_time = sc_time_stamp();
```

```
cout << "  The time is now "
          << sc_time_stamp()
          << "!" << endl;
```

Time

- method **sc_start**() is used to start simulation

```
//sim "forever"
sc_start();
//sim no more than max_sc_time
sc_start(const sc_time& max_sc_time);
//sim no more than max_time time_unit's
sc_start(double max_time, sc_time_unit time_unit);
```

```
//FILE: main.cpp
int sc_main(int argc, char* argv[]) { // args unused
  basic_process_ex my_instance("my_instance");
  sc_start(60.0,SC_SEC); // Limit sim to one minute
  return 0;
}
```

Time

- simulations use delays in simulated time to model
  - real world behaviors
  - mechanical actions
  - chemical reaction times
  - signal propagation
- **wait**() method provides a syntax to allow this delay in **SC_THREAD** processes

```
wait(delay_sc_time); // wait specified amount of
                     // time
```

Concurrency

- concurrency is fundamental to simulating with SystemC
- SystemC uses **simulation processes** to model concurrency
  - event-driven simulator
  - concurrency is not true concurrent execution
  - simulated concurrency works like cooperative multitasking
    - a simulation process runs: it is expected to execute a small segment of code and then return control to the simulation kernel

Concurrency

- SystemC simulation processe **(SC_THREAD)**
  - simply C++ function
    - designated by the programmer to be used as processes (**process registration**)
  - it can be used only within a SystemC module
    - the function must be a member function of the module class
  - must be used only during the elaboration stage
  - the member function must exist and the function can take no arguments and return no values

```
SC_THREAD(MEMBER_FUNCTION);
```

Concurrency

- processes must voluntarily yield control
  - executing a **return**
  - calling SystemC's **wait()** function
- processes typically begin execution at the start of simulation and continue in an endless loop until the simulation ends

```
//FILE: two_processes.h
SC_MODULE(two_processes) {
  void wiper_thread(void); // process
  void blinker_thread(void); // process
  SC_CTOR(two_processes) {
    SC_THREAD(wiper_thread); // register process
    SC_THREAD(blinker_thread); // register process
  }
};
```

Concurrency

```
//FILE: two_processes.cpp
void two_processes::wiper_thread(void) {
  while (true) {
    wipe_left();
    wait(500,SC_MS);
    wipe_right();
    wait(500,SC_MS);
  }//endwhile
}

void two_processes::blinker_thread(void) {
  while (true) {
    blinker = true;
    cout << "Blink ON" << endl;
    wait(300,SC_MS);
    cout << "Blink OFF" << endl;
    blinker = false;
    wait(300,SC_MS);
  }//endwhile
}
```

Concurrency

- **DEFINITION**: a SystemC event is the occurrence of an **sc_event** notification and happens at a single instant in time
- an event has no duration or value
- **RULE**: to observe an event, the observer must be watching for the event prior to its notification

```
sc_event event_name1[,event_name2]…;
```

```
event_name.notify(void);           // Immediate
event_name.notify(SC_ZERO_TIME);// Delayed
event_name.notify(sc_time);         // Timed (time>0)
event_name.notify(double,units);// Convenience
```

Concurrency

- events are explicitly caused using the notify() method of an sc_event object
- invoking an immediate **notify(void)** causes any processes waiting for the event to be immediately moved from the waiting set into the runnable set for execution

```
sc_event A_event;
A_event.notify(10,SC_NS);
A_event.notify( 5,SC_NS); // only this one stays
A_event.notify(15,SC_NS);
```

Concurrency

- thread processes rely on the **wait**() method to suspend their execution

```
wait(time);                   // timeout is the event
wait(double,time_unit);       // convenience
wait(event);                  // single event
wait(event1 | eventn…);       // any of these
wait(event1 & eventn…);       // all of these
wait(time,event);             // event or timeout
wait(time,event1 | eventn…);// any event or timeout
wait(time,event1 & eventn…);// all events or timeout
wait(); // static sensitivity - discussed later
```

Concurrency

```
…
sc_event ack_event, bus_error_event;
…
sc_time start_time(sc_time_stamp());
wait(t_MAX_DELAY, ack_event | bus_error_event);
if (sc_time_stamp()-start_time == t_MAX_DELAY) {
  break; // path for a time out
  …
```

Concurrency

- SystemC has more than one type of process
  - The **SC_METHOD** process is in some ways simpler than the **SC_THREAD**
  - **SC_METHOD** processes never suspend internally (i.e., they can never invoke **wait**())
  - **SC_METHOD** processes run completely and return

```
SC_METHOD(process_name);//Located INSIDE constructor
```

Concurrency

- **SC_METHOD** processes dynamically specify their sensitivity by means of the **next_trigger**() method

```
next_trigger(time);
next_trigger(timeout,time_unit); //convenience
next_trigger(event);
next_trigger(event1 | event2…); //any of these
next_trigger(event1 & event2…); //all of these
                                //required
next_trigger(timeout,event); //event with timeout
next_trigger(timeout,event1 | event2…);//any + timeout
next_trigger(timeout,event1 & event2…);//all + timeout
next_trigger(void); //re-establish static sensitivity
```

Concurrency

- The concept of actively determining what will cause a process to resume is often called **dynamic sensitivity**
- **Static sensitivity** establishes the parameters for resumption during elaboration (i.e., before simulation begins).
- Once established, static sensitivity parameters cannot be changed (i.e., they're static).

```
// IMPORTANT: Must follow process registration
sensitive << event [<< event]…; // streaming style
```

Concurrency

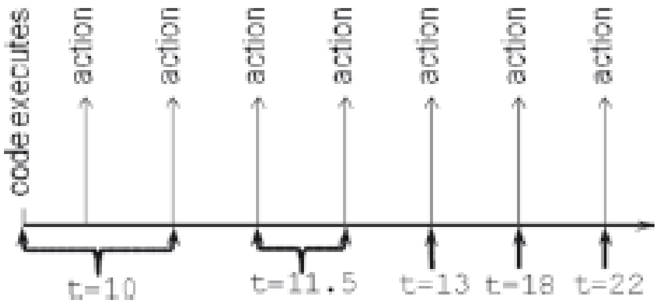
- the simulation engine description specifies that processes are executed at least once initially by placing processes in the runnable set during the initialization stage
- it may be necessary to specify that some processes should not be made runnable at initialization
  - SystemC provides the **dont_initialize**() method

```
...
SC_METHOD(attendant_method);
  sensitive(fillup_request);
  dont_initialize();
...
```

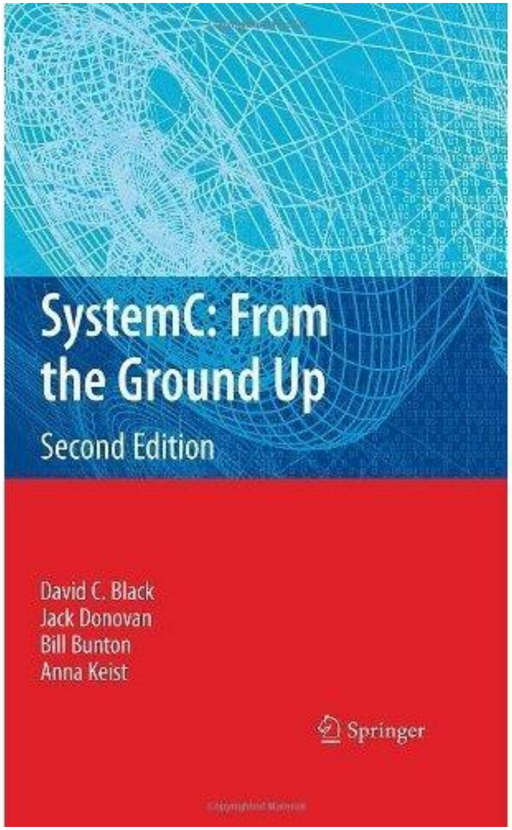
Concurrency

- **sc_event_queue**
  - allows a single event to be scheduled multiple times even for the same instant in time

```
sc_event_queue action;
wait(10,SC_NS)//assert time=10ns
sc_time now1(sc_time_stamp());//observe current time
action.notify(20,SC_NS);//schedule for 20ns from now
action.notify(10,SC_NS);//schedule for 20ns from now
action.cancel_all();//cancel all actions entirely
action.notify(8,SC_NS);//schedule for 8 ns from now
action.notify(1.5,SC_NS);// 1.5 ns from now
action.notify(1.5,SC_NS);// another identical action
action.notify(3.0,SC_NS);// 3.0 ns from now
action.notify(SC_ZERO_TIME);//after all runnable
action.notify(SC_ZERO_TIME);//and yet another
action.notify(12,SC_NS);// 12 ns from now
sc_time now2(sc_time_stamp());//observe current time
```



Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”

SISTEME DE CALCUL DEDICATE

Curs 4

Outline

- SystemC
  - Dynamic processes
  - Basic channels
  - Evaluate-update channels
- Bibliography

Dynamic processes

- SystemC 2.1 introduced the concept of ***dynamically spawned processes***
- useful in testbench scenarios
  - to track transaction completion
  - to spawn traffic generators dynamically

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
```

Dynamic processes

- declare the functions to be spawned as processes
- unlike static processes
  - dynamic processes may have up to eight arguments and a return value
    - the return value will be provided via a reference variable in the actual spawn function

```
// Ordinary function declarations
void inject(void); // no args or return
int count_changes(sc_signal<int>& sig);

// Method function declarations
class TestChan : public sc_module {
  …
  bool Track(sc_signal<packet>& pkt);
  void Errors(int maxwarn, int maxerr);
  void Speed(void);
  …
};
```

Dynamic processes

- define the implementation and register the function with the kernel
  - within an **SC_THREAD**
  - with restrictions within an **SC_METHOD**
- syntax to register dynamic processes with void return

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    sc_bind(&funcName, ARGS…)//no return value
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    sc_bind(&methName, object, ARGS…)//no return
   ,processName
   ,spawnOptions
  );
```

Dynamic processes

- syntax to register dynamic processes with return values
- object is a reference to the calling module
  - normally just use the C++ keyword **this**

```
sc_process_handle hname = // ordinary function
  sc_spawn(
    &returnVar
   ,sc_bind(&funcName, ARGS…)
   ,processName
   ,spawnOptions
  );

sc_process_handle hname = // member function
  sc_spawn(
    &returnVar
   ,sc_bind(&methodName, object, ARGS …)
   ,processName
   ,spawnOptions
  );
```

Dynamic processes

- by default, arguments are passed by value
- to pass by reference or by constant reference, a special syntax is required

```
sc_ref(var)   // reference
sc_cref(var) // constant reference
```

Dynamic processes

- spawn options are determined
  - by creating an **sc_spawn_option** object
  - invoking one of several methods that set the options

```
sc_spawn_option objname;
objname.spawn_method();// register as SC_METHOD
objname.dont_initialize();
objname.set_sensitivity(event_ptr);
objname.set_sensitivity(port_ptr);
objname.set_sensitivity(interface_ptr);
objname.set_sensitivity(event_finder_ptr);
objname.set_stack_size(value); // experts only!
```

Dynamic processes

```
#define SC_INCLUDE_DYNAMIC_PROCESSES
#include <systemc>
…
void spawned_thread() {// This will be spawned
  cout << "INFO: spawned_thread "
       << sc_get_current_process_handle().name()
       << " @ " << sc_time_stamp() << endl;
  wait(10,SC_NS);
  cout << "INFO: Exiting" << endl;
}

void simple_spawn::main_thread() {
  wait(15,SC_NS);
  // Unused handle discarded
  sc_spawn(sc_bind(&spawned_thread));
  cout << "INFO: main_thread " << name()
       << " @ " << sc_time_stamp() << endl;
  wait(15,SC_NS);
  cout << "INFO: main_thread stopping "
       << " @ " << sc_time_stamp() << endl;
}
```

Dynamic processes

```
// Add "& resume" to sensitivity while suspended
void sc_process_handle::suspend(descend);
void sc_process_handle::resume(descend);

// Ignore sensitivity while disabled
void sc_process_handle::disable(descend);
void sc_process_handle::enable(descend);

// Complete remove process
void sc_process_handle::kill(descend);

// Asynchronously restart a process
void sc_process_handle::reset(descend);

// Reset process on every resumption event
void sc_process_handle::sync_reset_on(descend);
void sc_process_handle::sync_reset_off(descend);

// Throw an exception in the specified process
template<typename T>
  void sc_process_handle::throw_it(
                            const T&,descend);
```

Basic channels

- communication of information between concurrent processes is done using
  - events
    - require careful coding
    - use handshake variable
  - ordinary module member data
  - channels - encapsulate complex communications
    - primitive
    - hierarchical

Basic channels

- primitive channels
  - inherit from the base class **sc_prim_channel**
  - also inherit from and implement one or more SystemC interface classes
  - types
    - **sc_mutex**
    - **sc_semaphore**
    - **sc_fifo**<*T*>

Basic channels

- mutex is short for ***mutually exclusive text***
- **sc_mutex** class
  - implements **sc_mutex_if** interface class
- blocking methods can only be used in **SC_THREAD** processes

```
sc_mutex NAME;

NAME.lock();       // Lock the mutex,
                   // wait until unlocked if in use
int NAME.trylock() // Non-blocking, returns success

NAME.unlock();     // Free a previously locked mutex
```

Basic channels

```
class bus : public sc_module {
  sc_mutex bus_access;
  …
  void write(int addr, int data) {
    bus_access.lock();
    // perform write
    bus_access.unlock();
  }
  …
};
```

```
void grab_bus_method() {
  if (bus_access.trylock() == 0) {
    // access bus
    …
    bus_access.unlock();
  }
}
```

Basic channels

- **sc_semaphore** class
  - inherits from and implements the **sc_semaphore_if** class

```
sc_semaphore NAME(COUNT);

NAME.wait();           // Lock one semaphore
                       // Wait until available if in use
int NAME.trywait()   // Non-blocking, return success

int NAME.get_value() // Returns available semaphores

NAME.post();           // Free one previously locked
                       // semaphore
```

Basic channels

```
class multiport_RAM {
  sc_semaphore read_ports(3);
  sc_semaphore write_ports(2);
  …
  void read(int addr, int& data) {
    read_ports.wait();
    // perform read
    read_ports.post();
  }
  void write(int addr, const int& data) {
    write_ports.wait();
    // perform write
    write_ports.post();
  }
  …
};//endclass
```

Basic channels

- the most popular channel for modeling at the architectural level is the **sc_fifo**<*T*> channel
  - inherits from and implements two interface classes:
    - **sc_fifo_in_if**<*T*>
    - **sc_fifo_out_if**<*T*>

Basic channels

```
sc_fifo<ELEMENT_TYPENAME> NAME(SIZE);

NAME.write(VALUE);
NAME.read(REFERENCE);
… = NAME.read() /* function style */
if (NAME.nb_read(REFERENCE)) { // Non-blocking
                               // true if success
  …
}
if (NAME.num_available() == 0)
  wait(NAME.data_written_event());
if (NAME.num_free() == 0)
  next_trigger(NAME.data_read_event());
```

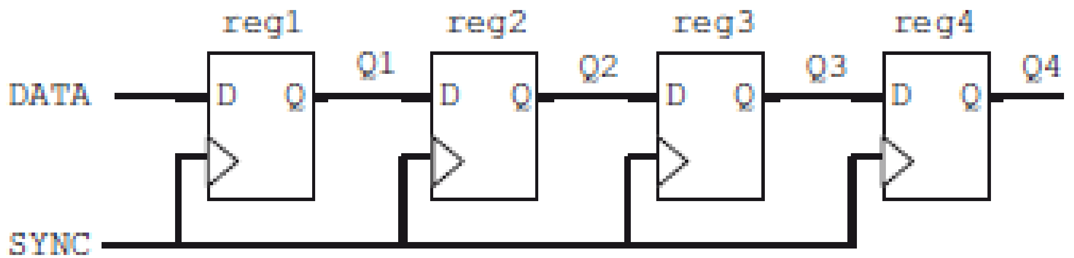
Basic channels

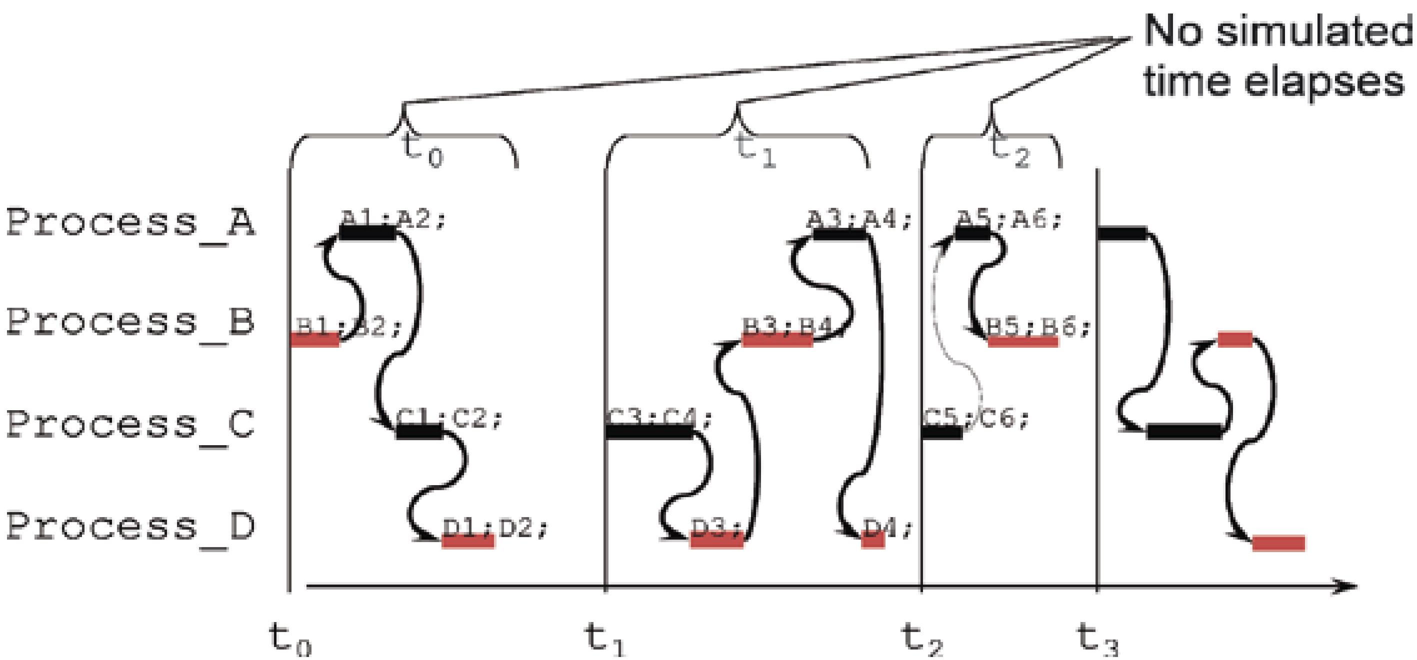
```
SC_MODULE(kahn_ex) {
  …
  sc_fifo<double> a, b, y;
  …
};
// Constructor
kahn_ex::kahn_ex() : a(24), b(24), y(48)
{
  …
}
void kahn_ex ::stim_thread() {
  for (int i=0; i!=1024; ++i) {
    a.write(double(rand()/1000));
    b.write(double(rand()/1000));
  }
}
void kahn_ex::addsub_thread() {
  while(true) {
    y.write(kA*a.read() + kB*b.read());
    y.write(kA*a.read() - kB*b.read());
  }//endforever
}
void kahn_ex::monitor_method() {
  cout << y.read() << endl;
}
```

Evaluate-Update Channels

- electronic hardware
  - behave in a manner approaching instantaneous activity
  - electronic signals have
    - a single source (producer)
    - multiple sinks (consumer)
      - it is quite important that all sinks “see” a signal update at the same time

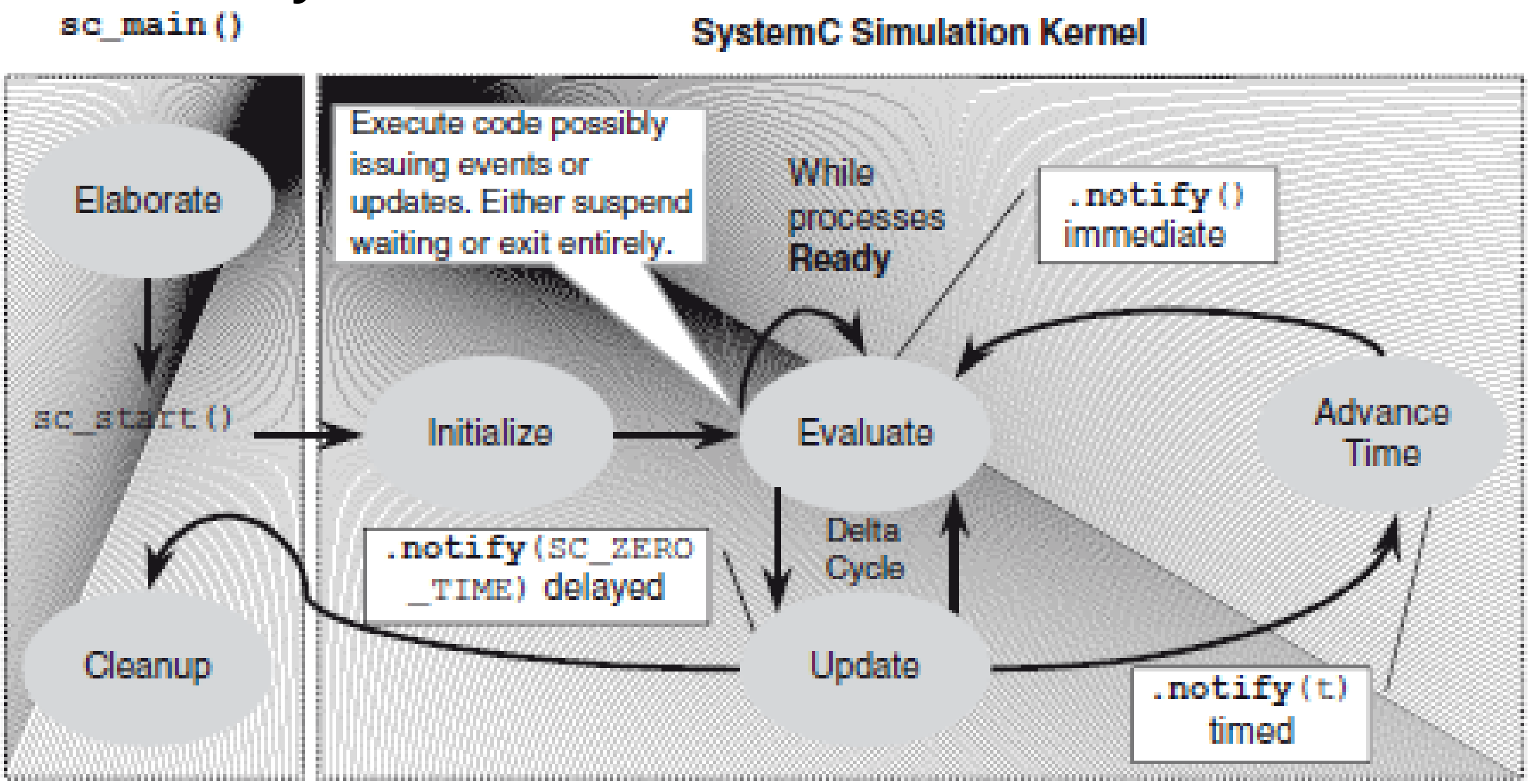
Evaluate-Update Channels





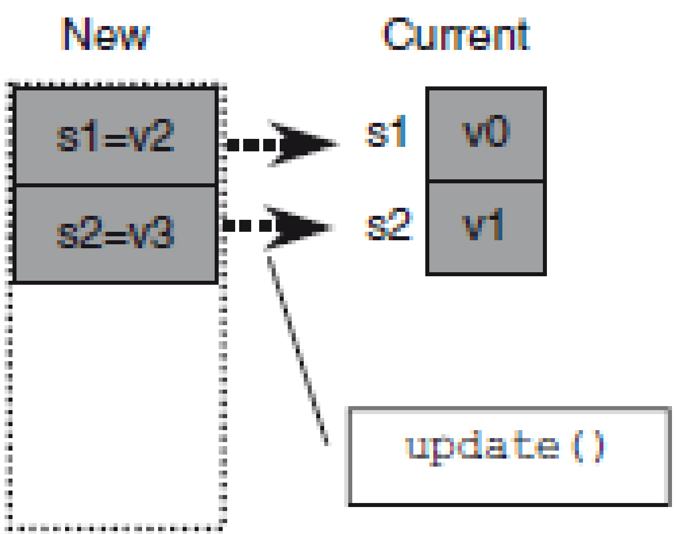
Evaluate-Update Channels

- evaluate-update paradigm
  - delta-cycle



Evaluate-Update Channels

- signal channels, use update phase as a point of data synchronization
- to accomplish this synchronization, every channel has two storage locations:
  - the current value and the new value



Evaluate-Update Channels

- **sc_signal**<*T*> primitive channel and its close relative, **sc_buffer**<*T*> both use the evaluate-update paradigm

```
  sc_signal<datatype> signame[, signame₁]…;//define
…
  signame.write(newvalue);
  varname  = signame.read();
  wait(signame.value_changed_event()|...);
  wait(signame.default_event()|...);
  if (signame.event() == true) {
    // occurred in previous delta-cycle
```

```
// Declare variables
int                  count;
string               message_temp;
sc_signal<int>       count_sig;
sc_signal<string> message_sig;

cout << "Initialize during 1st delta cycle" << endl;
count_sig.write(10);
message_sig.write("Hello");
count = 11;
message_temp = "Whoa";
cout << "count is " << count << " "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "' "
     << "message_sig is '" << message_sig << "'"
     << endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "2nd delta cycle" << endl;
count = 20;
count_sig.write(count);
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     <<endl << "Waiting" << endl << endl;
wait(SC_ZERO_TIME);

cout << "3rd delta cycle" << endl;
message_sig.write(message_temp = "Rev engines");
cout << "count is " << count << ", "
     << "count_sig is " << count_sig << endl
     << "message_temp is '" << message_temp << "', "
     << "message_sig is '" << message_sig << "'"
     << endl << endl << "Done" << endl;
```

```
Initialize during 1st delta cycle
count is 11, count_sig is 0
message_temp is 'Whoa', message_sig is ''
Waiting

2nd delta cycle
count is 20, count_sig is 10
message_temp is 'Whoa', message_sig is 'Hello'
Waiting

3rd delta cycle
count is 20, count_sig is 20
message_temp is 'Rev engines', message_sig is
'Hello'

Done
```

Evaluate-Update Channels

- multiple writers

```
sc_signal_resolved name;
sc_signal_rv<WIDTH> name;
```

Multiple Drivers on a Bus

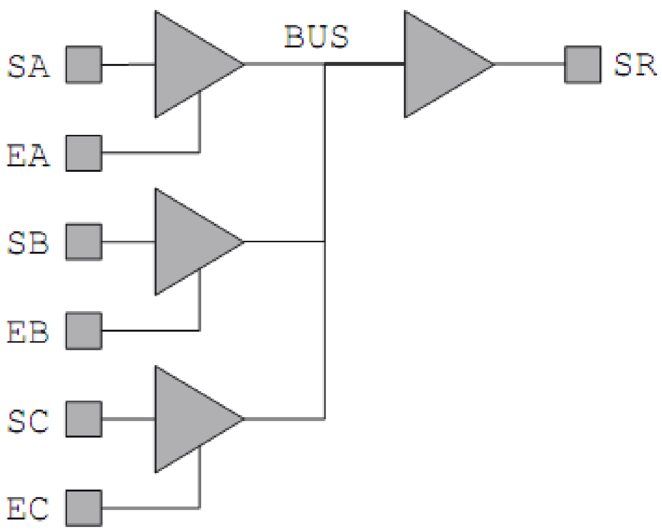


Table 9.1 Resolution functionality for `sc_signal_resolved`

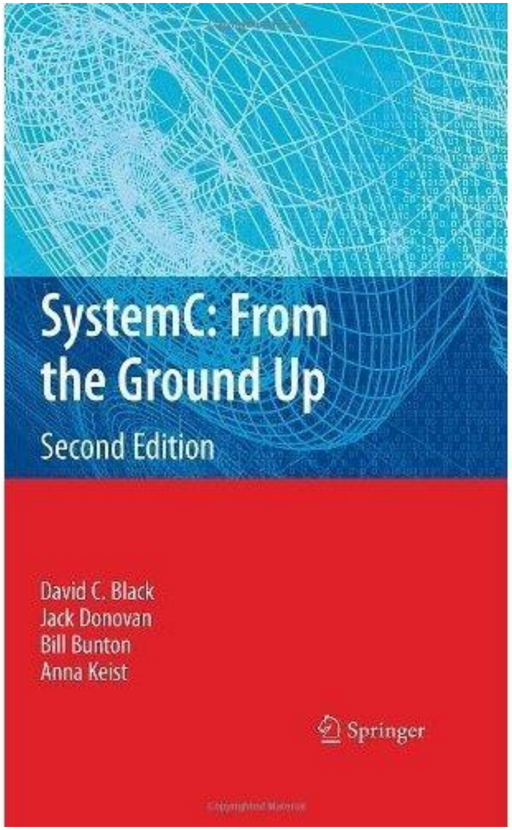
A\B	'0'	'1'	'X'	'Z'
'0'	'0'	'X'	'X'	'0'
'1'	'X'	'1'	'X'	'1'
'X'	'X'	'X'	'X'	'X'
'Z'	'0'	'1'	'X'	'Z'

Evaluate-Update Channels

- template specializations
  - a template specialization occurs when a definition is provided for a specific template value
  - specialized templates
    - sc_signal<bool>
    - sc_signal<sc_logic>

```
sensitive << signame.posedge_event()
          << signame.negedge_event();
wait(signame.posedge_event()
    |signame.negedge_event());
if (signame.posedge_event()
   |signame.negedge_event()) {
```

Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”

SISTEME DE CALCUL DEDICATE

Curs 5

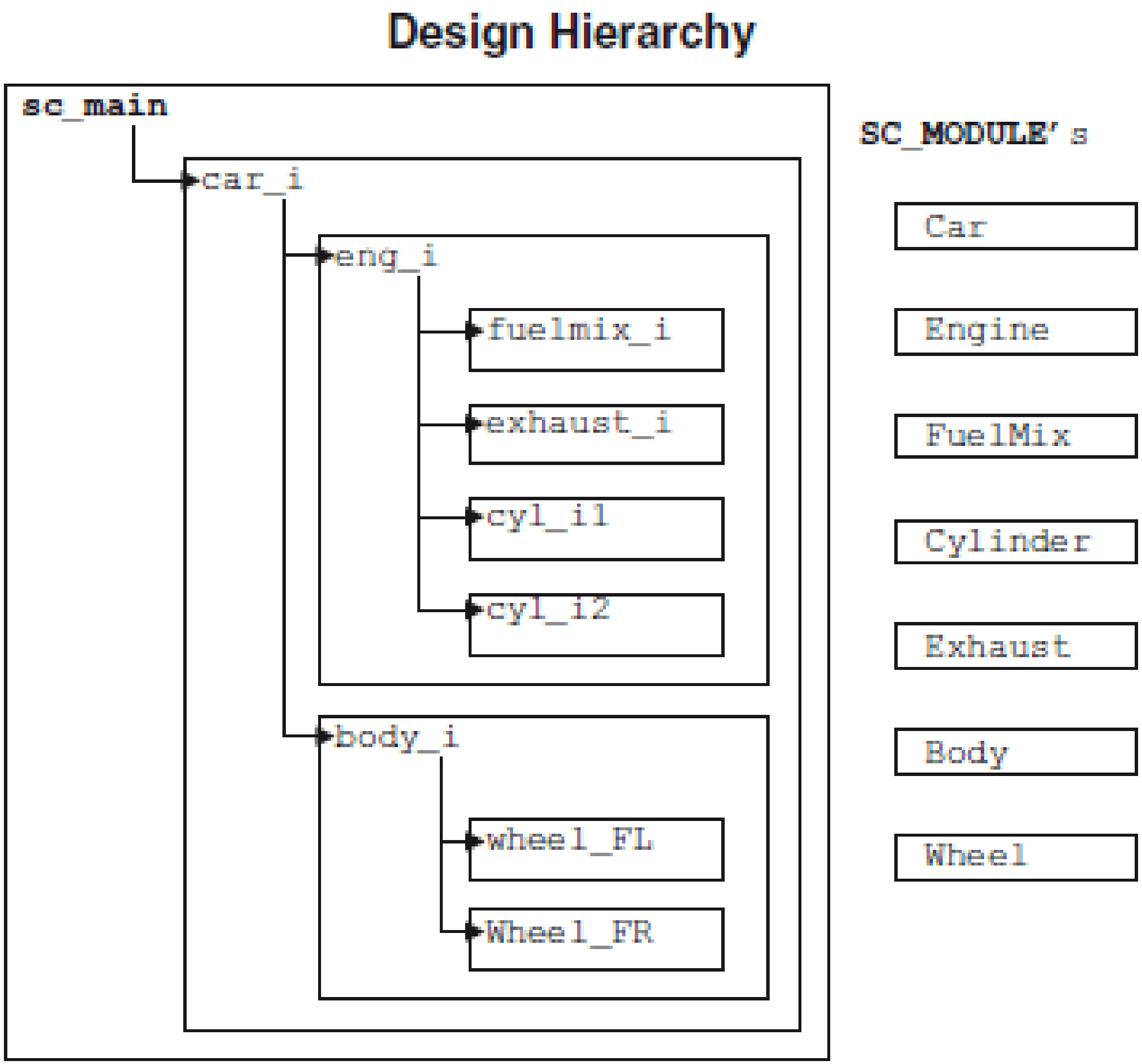
Outline

- SystemC
  - Design hierarchy
  - Ports
- Bibliography

Design hierarchy

- design hierarchy
  - hierarchical relationships of modules
  - connectivity that lets modules communicate in an orderly fashion
  - in SystemC uses instantiations of modules as member data of parent modules
    - to create a level of hierarchy, create an **sc_module** object within a parent **sc_module**.

Design hierarchy



Design hierarchy

- C++ offers two basic ways to create submodule objects
  - a submodule object may be created ***directly*** by declaration
  - a submodule object may be ***indirectly*** referenced by means of a pointer in combination with dynamic allocation

Design hierarchy

- six approaches
  - Direct top-level (**sc_main**)
  - Indirect top-level (**sc_main**)
  - Direct submodule header-only
  - Direct submodule
  - Indirect submodule header-only
  - Indirect submodule

Design hierarchy

- top-level implementation with direct instantiation

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car car_i("car_i");
    sc_start();
    return 0;
  }
```

Design hierarchy

- top-level implementation with indirect instantiation

```
//FILE: main.cpp
  #include <systemc>
  #include "Car.h"
  int sc_main(int argc, char* argv[]) {
    Car* car_iptr;                   // pointer to Car
    car_iptr = new Car("car_i"); // create Car
    sc_start();
    delete car_iptr;
    return 0;
  }
```

Design hierarchy

- direct instantiation in the header
  - use of an initializer list

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i ;
    SC_CTOR(Car)
    : body_i("body_i") //initialization
    , eng_i("eng_i")   //initialization
    {
      // other initialization
    }
  };
```

Design hierarchy

- direct instantiation and separate compilation

```
//FILE:Car.h
  #include "Body.h"
  #include "Engine.h"
  SC_MODULE(Car) {
    Body   body_i;
    Engine eng_i;
    Car(sc_module_name nm);
  };
```

```
//FILE:Car.cpp
  #include <systemc>
  #include "Car.h"
  // Constructor
  SC_HAS_PROCESS(Car);
  Car::Car(sc_module_name nm)
  : sc_module(nm)
  , body_i("body_i")
  , eng_i("eng_i")
  {
    // other initialization
  }
```

Design hierarchy

- Indirect Submodule Header-Only Implementation

```
//FILE:Body.h
  #include "Wheel.h"
  SC_MODULE(Body) {
    Wheel* wheel_FL_iptr;
    Wheel* wheel_FR_iptr;
    SC_CTOR(Body) {
      wheel_FL_iptr = new Wheel("wheel_FL_i");
      wheel_FR_iptr = new Wheel("wheel_FR_i");
      // other initialization
    }
  };
```

Design hierarchy

- Indirect Submodule Implementation

```
//FILE:Engine.h
  class FuelMix;
  class Exhaust;
  class Cylinder;
  SC_MODULE(Engine) {
    FuelMix*   fuelmix_iptr;
    Exhaust*   exhaust_iptr;
    Cylinder*  cyl1_iptr;
    Cylinder*  cyl2_iptr;
    Engine(sc_module_name nm); // Constructor
  };
```

Design hierarchy

- Indirect Submodule Implementation
  - good for IP distribution

```
//FILE: Engine.cpp
  #include <systemc>
  #include "FuelMix.h"
  #include "Exhaust.h"
  #include "Cylinder.h"
  // Constructor
  SC_HAS_PROCESS(Engine);
  Engine::Engine(sc_module_name nm)
  : sc_module(nm)
  {
    fuelmix_iptr = new FuelMix("fuelmix_i");
    exhaust_iptr = new Exhaust("exhaust_i");
    cyl1_iptr    = new Cylinder("cyl1_i");
    cyl2_iptr    = new Cylinder("cyl2_i");
    // other initialization
  }
```

Design hierarchy

Level	Allocation	Pros	Cons
Main	Direct	Least code	Inconsistent with other levels
Main	Indirect	Dynamically configurable	Involves pointers
Module	Direct header only	All in one file Easier to understand	Requires submodule headers
Module	Indirect header only	All in one file Dynamically configurable	Involves pointers Requires submodule headers
Module	Direct with separate compilation	Hides implementation	Requires submodule headers
Module	Indirect with separate compilation	Hides submodule headers and implementation Dynamically configurable	Involves pointers

Ports

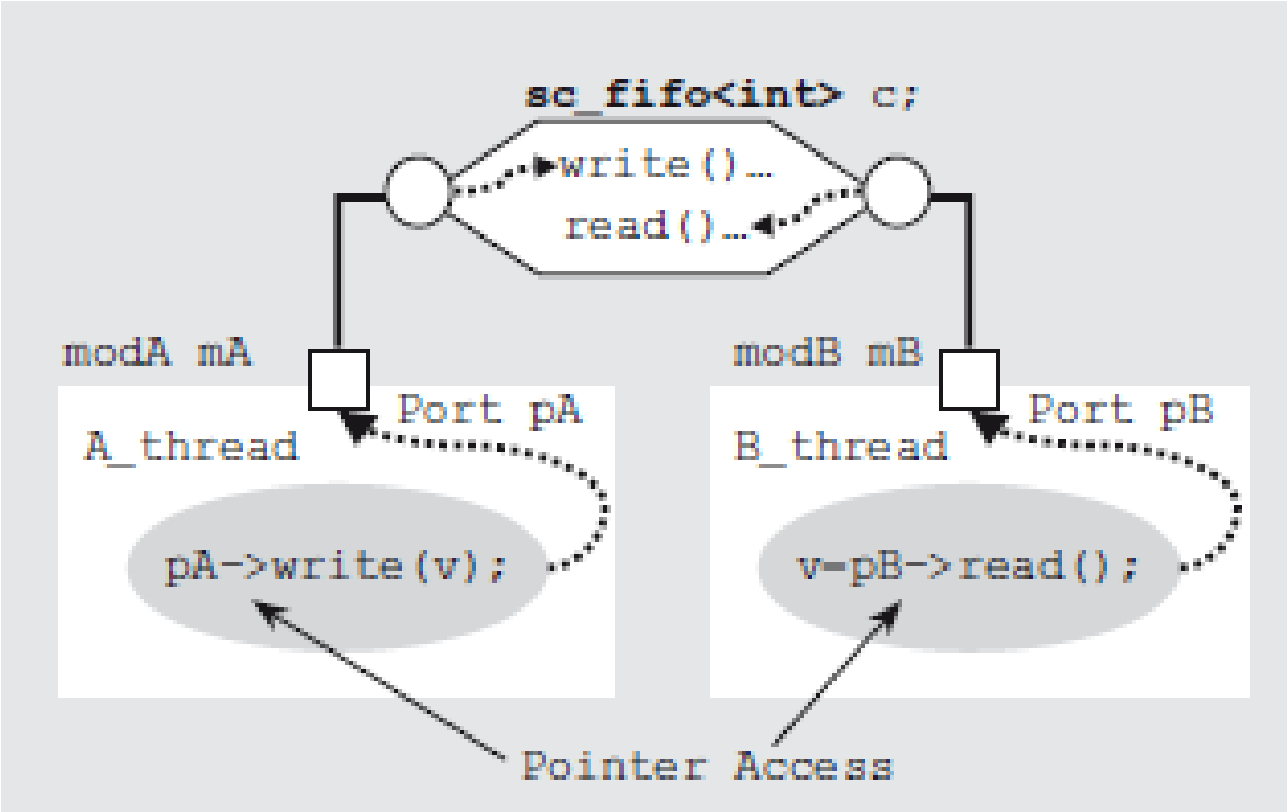
- what is the best way to communicate?
  - safety
    - is a concern because all activity occurs within processes
    - care must be taken when communicating between processes to avoid race conditions.
    - events and channels are used to handle this concern

Ports

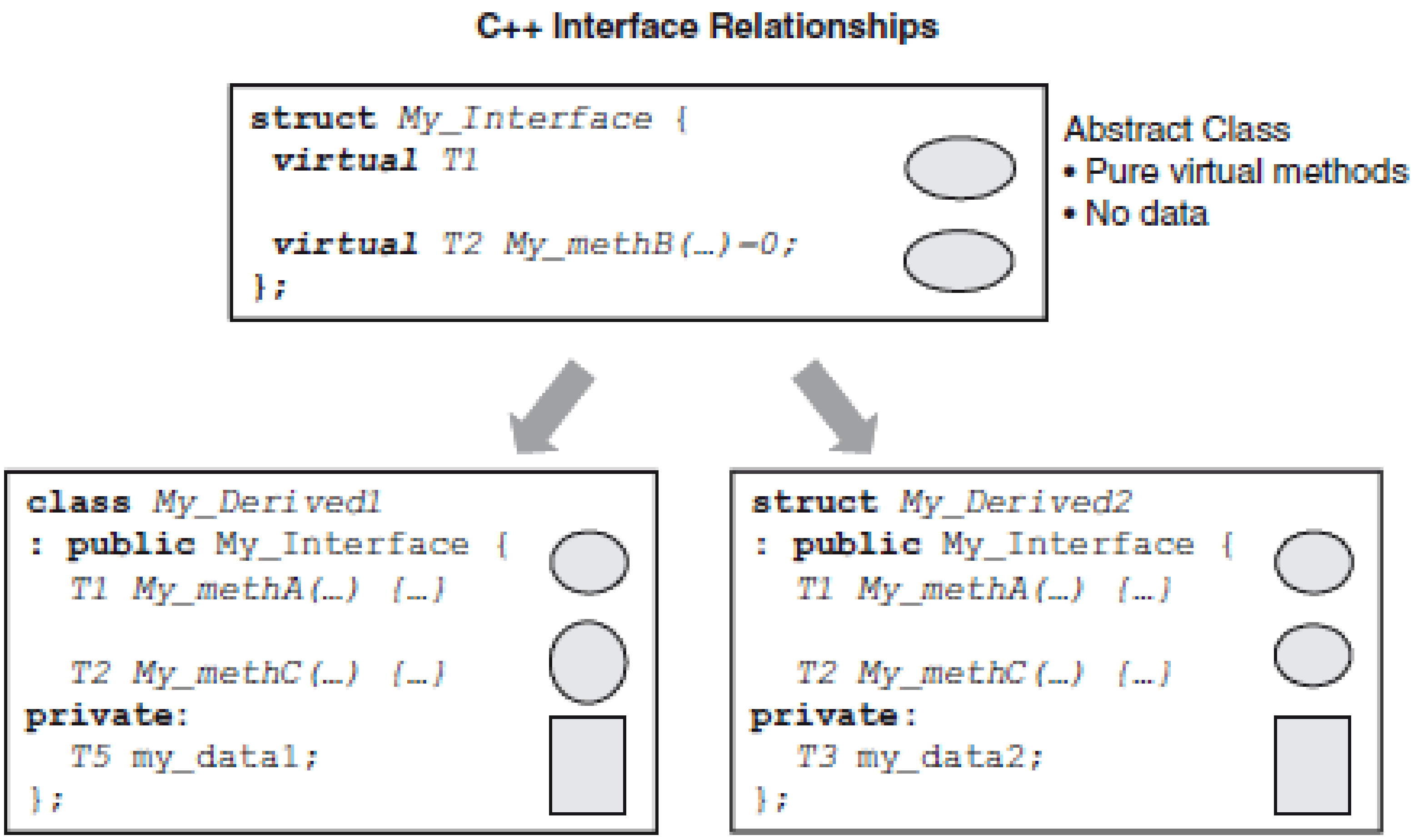
- what is the best way to communicate?
  - easy of use
    - dispense with any solution involving global variables
    - a process that monitors and manages events defined in instantiated modules (awkward)
    - SystemC takes an approach that lets modules use channels inserted between the communicating modules
    - a concept called a port
      - a pointer to a channel outside the module

Ports

Communication Via sc_ports



Ports



C++ Interface Relationships
struct My_Interface {
virtual T1
virtual T2 My_methB(…)=0;
};
Abstract Class
• Pure virtual methods
• No data
class My_Derived1
: public My_Interface {
T1 My_methA(…) {…}
T2 My_methC(…) {…}
private:
T5 my_data1;
};
struct My_Derived2
: public My_Interface {
T1 My_methA(…) {…}
T2 My_methC(…) {…}
private:
T3 my_data2;
};

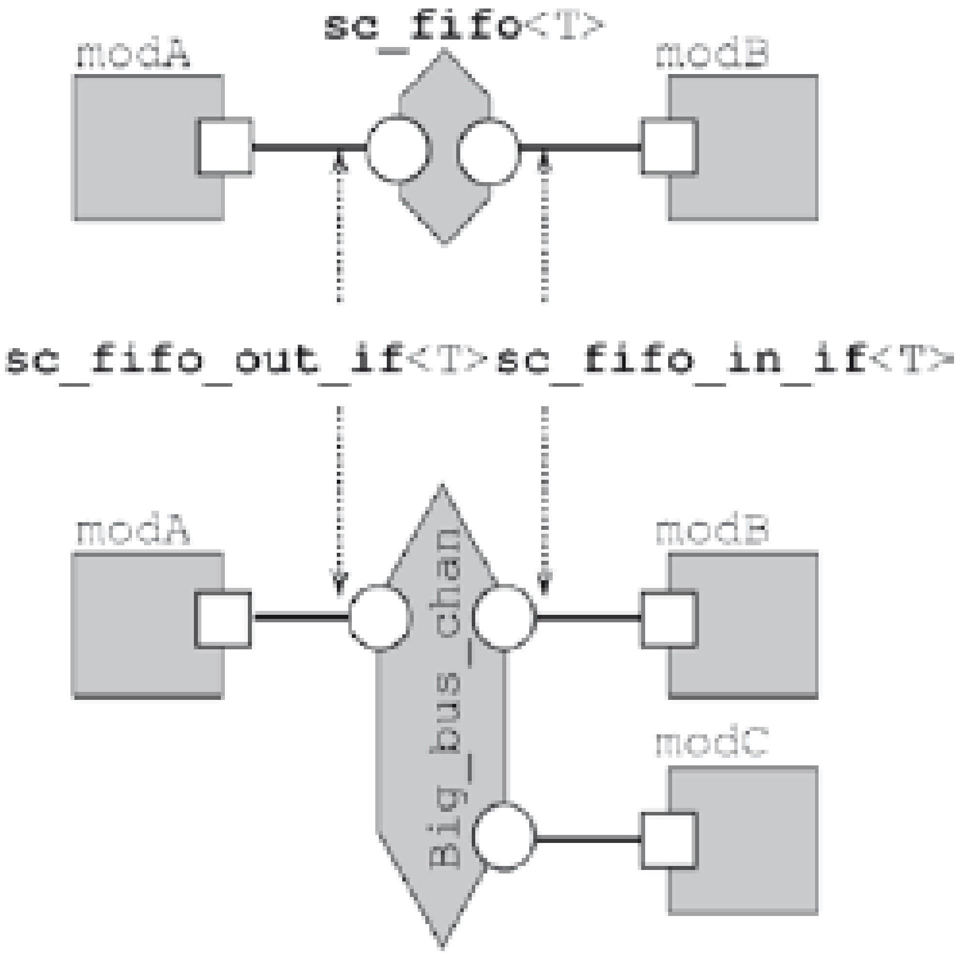
Ports

- **DEFINITION**: A SystemC interface is an abstract class that inherits from **sc_interface** and provides only pure virtual declarations of methods referenced by SystemC channels and ports. No implementations or data are provided in a SystemC interface.

Ports

- **DEFINITION**: A SystemC channel is a class that inherits from either **sc_channel** or from **sc_prim_channel**, and the channel should1 inherit and implement one or more SystemC interface classes. A channel implements all the pure virtual methods of the inherited interface classes.

Ports


sc_fifo<T>
modA
modB
sc_fifo_out_if<T>
sc_fifo_in_if<T>
modA
Big_bus_chan
modB
modC

Ports

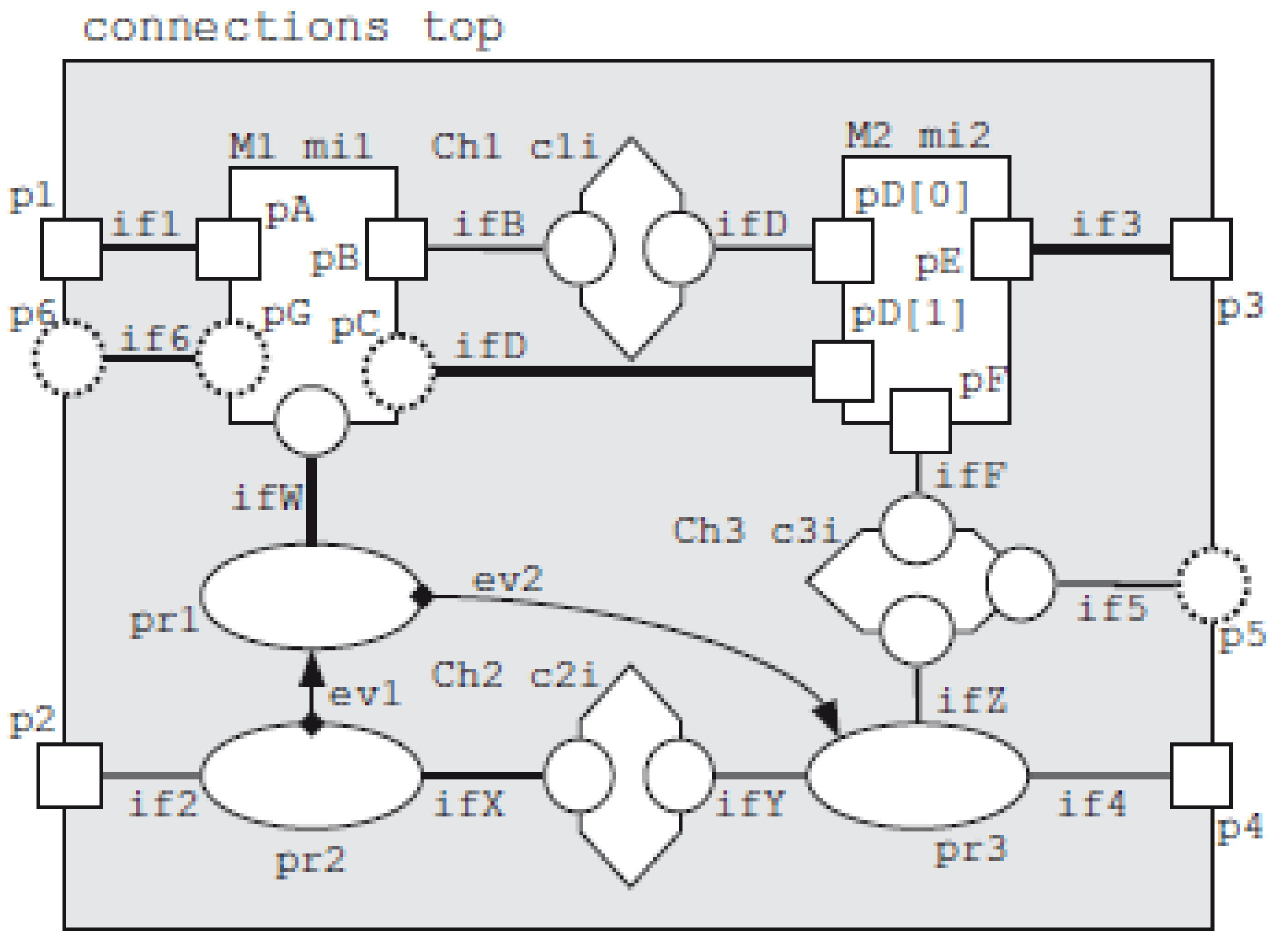
- **DEFINITION** A SystemC port is a class templated with and inheriting from a SystemC interface. Ports allow access of channels across module boundaries.

```
sc_port<interface> portname;
```

```
SC_MODULE(stereo_amp) {
  sc_port<sc_fifo_in_if<int> >  soundin_p;
  sc_port<sc_fifo_out_if<int> > soundout_p;
  …
};
```

Ports

Port Connections



Ports

- modules are connected to channels after both the modules and channels have been instantiated
- two syntaxes for connecting ports
  - by name
  - by position

```
mod_inst.portname(channel_instance); // Named
mod_instance(channel_instance,…); // Positional
```

Ports

- When the code instantiating an **sc_port** executes:
  - the **operator()** is overloaded to take a channel object by reference
  - saves a pointer to that reference internally for later access by the port
- a port is an interface pointer to a channel that implements the interface

Ports

```
//FILE: Rgb2YCrCb.h
SC_MODULE(Rgb2YCrCb) {
  sc_port<sc_fifo_in_if<RGB_frame> >     rgb_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > ycrcb_po;
};
```

```
//FILE: YCRCB_Mixer.h
SC_MODULE(YCRCB_Mixer) {
  sc_port<sc_fifo_in_if<float> >         K_pi;
  sc_port<sc_fifo_in_if<YCRCB_frame> >  a_pi, b_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > y_po;
};
```

Ports

```
//FILE: VIDEO_Mixer.h
SC_MODULE(VIDEO_Mixer) {
  // ports
  sc_port<sc_fifo_in_if<YCRCB_frame> >  dvd_pi;
  sc_port<sc_fifo_out_if<YCRCB_frame> > video_po;
  sc_port<sc_fifo_in_if<MIXER_ctrl> >   control;
  sc_port<sc_fifo_out_if<MIXER_state> > status;
  // local channels
  sc_fifo<float>       K;
  sc_fifo<RGB_frame>   rgb_graphics;
  sc_fifo<YCRCB_frame> ycrcb_graphics;
  // local modules
  Rgb2YCrCb   Rgb2YCrCb_i;
  YCRCB_Mixer YCRCB_Mixer_i;
  // constructor
  VIDEO_Mixer(sc_module_name nm);
  void Mixer_thread();
};
```

Ports

```
SC_HAS_PROCESS(VIDEO_Mixer);
VIDEO_Mixer::VIDEO_Mixer(sc_module_name nm)
: sc_module(nm)
, Rgb2YCrCb_i("Rgb2YCrCb_i")
, YCRCB_Mixer_i("YCRCB_Mixer_i")
{
  // Connect
  Rgb2YCrCb_i.rgb_pi(rgb_graphics);
  Rgb2YCrCb_i.ycrcb_po(ycrcb_graphics);
  YCRCB_Mixer_i.K_pi(K);
  YCRCB_Mixer_i.a_pi(dvd_pi);
  YCRCB_Mixer_i.b_pi(ycrcb_graphics);
  YCRCB_Mixer_i.y_po(video_po);
}
```

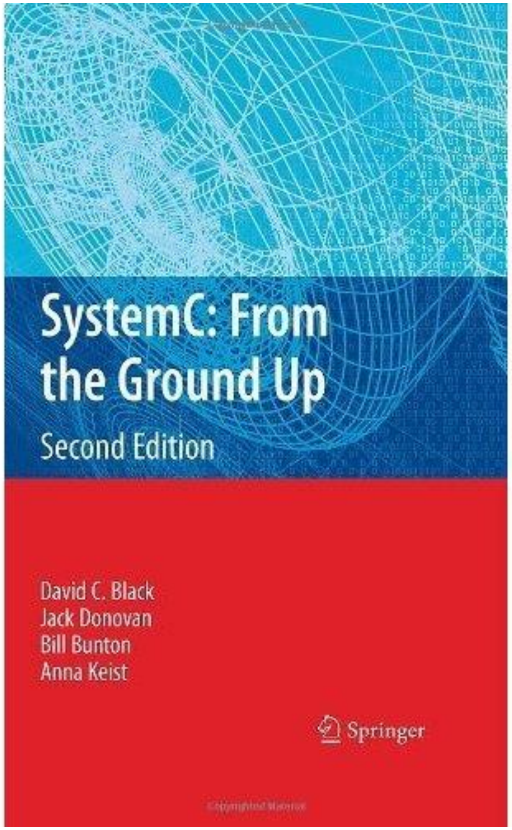
Ports

- the sc_port overloads the C++ **operator->()**, which allows a simple syntax

```
portname->method(optional_args);
```

```
void VIDEO_Mixer::Mixer_thread() {
  …
  switch (control->read()) {
    case MOVIE: K.write(0.0f); break;
    case MENU:  K.write(1.0f); break;
    case FADE:  K.write(0.5f); break;
    default:    status->write(ERROR); break;
  }
  …
}
```

Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”

SISTEME DE CALCUL DEDICATE

Curs 6

Outline

- SystemC
  - Specialized ports, sc_export
- Bibliography

Specialized ports, sc_export

- SystemC provides a variety of standard interfaces that go hand in hand with the built-in channels
  - basis for creating custom channels

Specialized ports, sc_export

- SystemC FIFO Interfaces for the **sc_fifo**<*T*>channel
  - **sc_fifo_in_if**<*T*>
  - **sc_fifo_out_if**<*T*>
    - provide all of the methods implemented by **sc_fifo**<*T*>
- the interfaces were defined prior to the creation of the channel
- the channel simply becomes the place to implement the interfaces and holds the data implied by the functionality of a FIFO

Specialized ports, sc_export

```
// Definition of sc_fifo<T> output interface
template <class T>
class sc_fifo_out_if: virtual public sc_interface {
public:
  virtual void write(const T& ) = 0;
  virtual bool nb_write(const T& ) = 0;
  virtual int num_free() const = 0;
  virtual const sc_event&
                    data_read_event() const = 0;
};
```

```
// Definition of sc_fifo<T> input interface
template<class T>
class sc_fifo_in_if: virtual public sc_interface{
public:
  virtual void read( T& ) = 0;
  virtual T read() = 0;
  virtual bool nb_read( T& ) = 0;
  virtual int num_available()const = 0;
  virtual const sc_event&
              data_written_event() const = 0;
};
```

Specialized ports, sc_export

- SystemC Signal Interfaces for the sc_signal<T>channel
  - **Sc_signal_in_if**<*T*>
  - **Sc_signal_inout_if**<*T*>
    - Provide all of the methods provided by **sc_signal**<*T*>

Specialized ports, sc_export

```
// Definition of sc_signal<T> input/output interface
template<class T>
class sc_signal_inout_if: public sc_signal_in_if<T>
{
public:
  virtual void write( const T& ) = 0;
};
```

```
// Definition of sc_signal<T> input interface
template<class T>
class sc_signal_in_if: virtual public sc_interface {
public:
  virtual const sc_event&
                value_changed_event() const = 0;
  virtual const T& read() const = 0;
  virtual bool event() const = 0;
};
```

Specialized ports, sc_export

- sc_mutex and sc_semaphore interfaces

```
// Definition of sc_mutex_if interface
class sc_mutex_if: virtual public sc_interface {
public:
  virtual int lock() = 0;
  virtual int trylock() = 0;
  virtual int unlock() = 0;
};
```

```
// Definition of sc_semaphore_if interface
class sc_semaphore_if: virtual public sc_interface
{
public:
  virtual int wait() = 0;
  virtual int trywait() = 0;
  virtual int post() = 0;
  virtual int get_value() const = 0;
};
```

Specialized ports, sc_export

- ports are defined on interfaces to channels
  - allow sensitivity to events defined on those channels
    - Example: process statically sensitive to the data_written_event()
    - Example: monitor an sc_signal<T> for any change in the data using the value_changed_event()
- ***Problem***: Ports are pointers that become initialized during elaboration, and they are undefined at the time when the **sensitive** method needs to know about them

Specialized ports, sc_export

- solution: a special class **sc_event_finder**
  - defers the determination of the actual event until after elaboration
  - an **sc_event_finder** must be defined for each event defined by the interface

Specialized ports, sc_export

```
classeslx_port
   :public sc_port<sc_signal_in_if<bool>,1>
 {
public:
// Use a typedef to shorten syntax below
   typedef sc_signal_in_if<bool>if_type;
   sc_event_finder& ef_posedge_event() const {
     return *newsc_event_finder_t<if_type>(
                              *this,
                              &if_type::posedge_event
                            );
   }//end ef_posedge_event
 };
```

```
SC_MODULE(my_module) {
   eslx_port my_p;
   …
   SC_CTOR(…) {
     SC_METHOD(my_method);
     sensitive<< my_p.ef_posedge_event();
   }
   void my_method();
   …
 };
```

Specialized ports, sc_export

- SystemC provides a set of template specializations that provide port definitions on the standard interfaces and include the appropriate event finders

Specialized ports, sc_export

```
// sc_port<sc_fifo_in_if<T>>
sc_fifo_in<T> name_fifo_ip;
sensitive<<name_fifo_ip.data_written();
value = name_fifo_ip.read();
name_fifo_ip.read(value);
if (name_fifo_ip.nb_read(value))...
if (name_fifo_ip.num_available())...
wait(name_fifo_ip.data_written_event());

// sc_port<sc_fifo_out_if<T>>
sc_fifo_out<T> name_fifo_op;
sensitive<<name_fifo_op.data_read();
name_fifo_op.write(value);
if (name_fifo_op.nb_write(value))...
if (name_fifo_op.num_free())...
wait(name_fifo_op.data_read_event());
```

Don't use dot (.) Use arrow (->) syntax.

GUIDELINE: Use dot (.) in the elaboration section of the code, but use arrow (->) in processes.

Specialized ports, sc_export

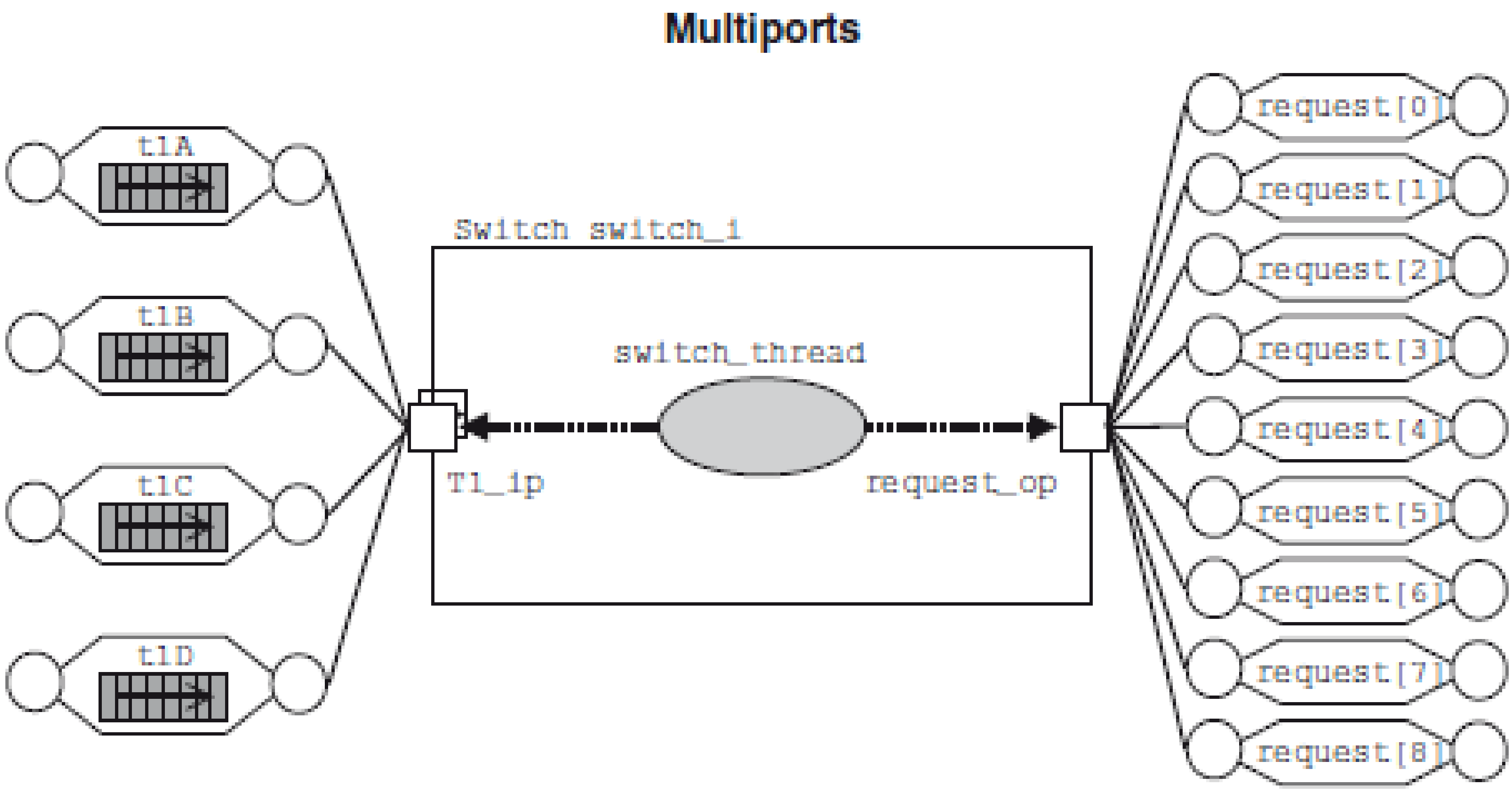
```
// sc_port<sc_signal_in_if<T>>
 sc_in<T> name_sig_ip;
 sensitive << name_sig_ip.value_changed();

 // Additional sc_in specializations...
 sc_in<bool> name_bool_sig_ip;
 sc_in<sc_logic >name_log_sig_ip;
 sensitive << name_sig_ip pos();
 sensitive << name_sig_ip neg();

 // sc_port<sc_signal_out_if<T>>
 sc_inout<T> name_sig_op;
 sensitive << name_sig_op.value_changed();
 sc_inout_resolved<N> name_rsig_op;
 sc_inout_rv<N> name_rsig_op;
 sc_inout<T> name_rsig_op;
 sc_inout_resolved<T> name_rsig_op;
 sc_inout_rv<T> name_rsig_op;
 // everything under sc_in<T> plus the following...
 name_sig_op.initialize(value);
 name_sig_op = value; // <-- DON'T USE!!!
```

Specialized ports, sc_export

- the **sc_port**<*T*> provides additional template parameters:
  - the array size parameter
    - multi-port or port array
  - the port policy parameter

```
sc_port<interface[,N[,POL]]> portname;
// N=0..MAX Default N=1
// POL is of type sc_port_policy
// POL defaults to SC_ONE_OR_MORE_BOUND
```

Specialized ports, sc_export



Specialized ports, sc_export

```
//FILE: Switch.h
  SC_MODULE(Switch) {
    sc_port<sc_fifo_in_if<int>
            ,5
            ,SC_ONE_OR_MORE_BOUND
            > T1_ip;
    sc_port<sc_signal_inout_if<bool>
            ,0
            > request_op;
    ...
  };
```

Specialized ports, sc_export

```
//FILE: Board.h
#include "Switch.h"
SC_MODULE(Board) {
  Switch switch_i;
sc_fifo<int> t1A, t1B, t1C, t1D;
sc_signal bool> request[9];
  SC_CTOR(Board): switch_i("switch_i")
  {
    // Connect 4 T1 channels to the switch
    switch_i.T1_ip(t1A);
    switch_i.T1_ip(t1B);
    switch_i.T1_ip(t1C);
    switch_i.T1_ip(t1D);
    // Connect 9 request channels to the
    // switch request output ports
    for (unsigned i=0;i!=9;i++) {
      switch_i.request_op(request[i]);
    }//endfor
    ...
  }//end constructor
  ...
};
```

From preceding example.

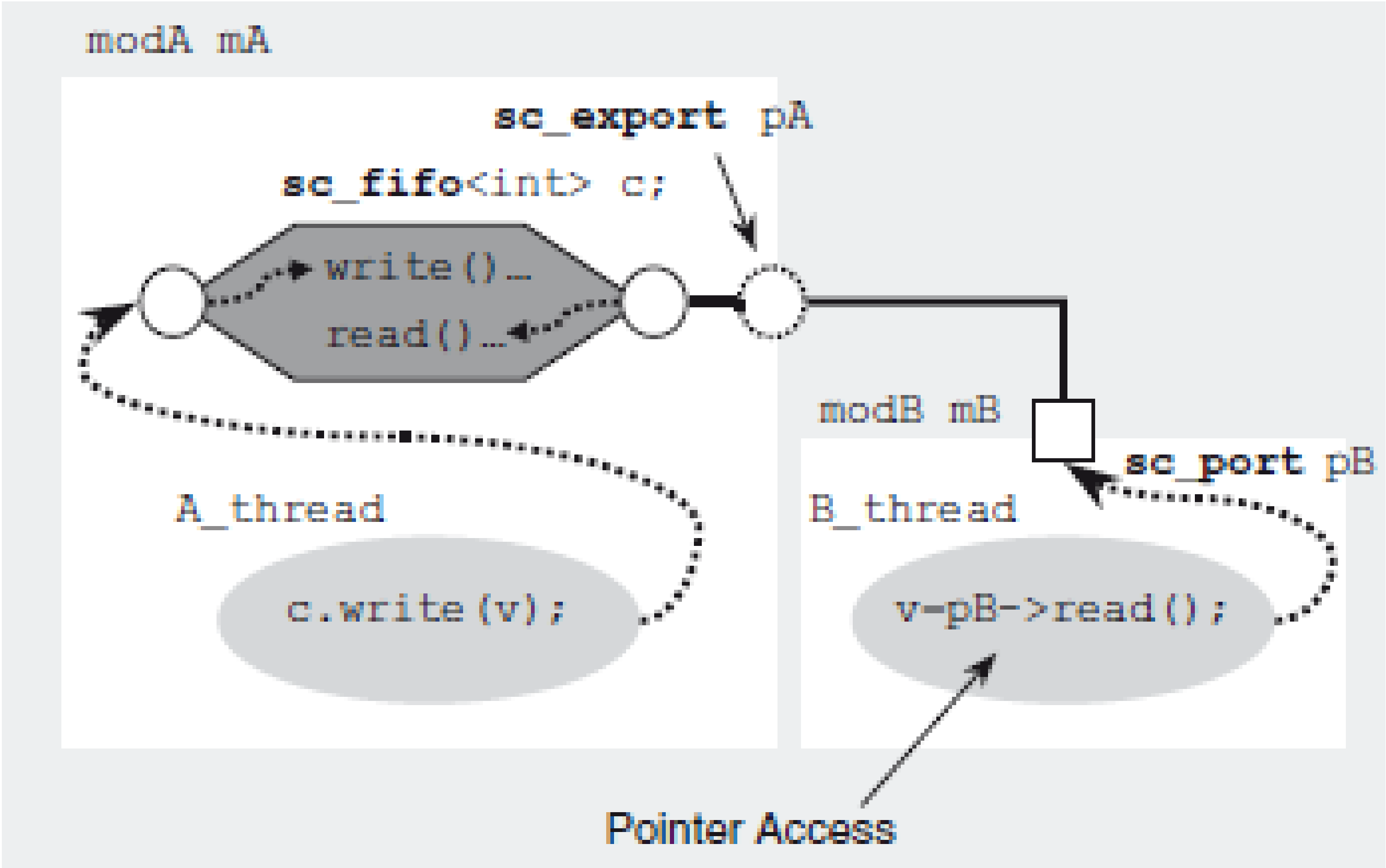
Specialized ports, sc_export

```
//FILE: Switch.cpp
void Switch::switch_thread() {
  // Initialize requests
  for (unsigned i=0;i!=request_op.size();i++) {
    request_op[i]->write(true);
  }//endfor
  // Startup after first port is activated
  wait(T1_ip[0]->data_written_event()
      |T1_ip[1]->data_written_event()
      |T1_ip[2]->data_written_event()
      |T1_ip[3]->data_written_event()
  );
  while(true) {
    for (unsigned i=0;i!=T1_ip.size();i++) {
      // Process each port...
      int value = T1_ip[i]->read();
    }//endfor
  }//endwhile
}//end Switch::switch_thread
```

Specialized ports, sc_export

- there is a second type of port called the **sc_export**<*T*>
  - differs in connectivity
  - the idea of an **sc_export**<*T*> is to move the channel inside the defining module
    - hide some of the connectivity details
    - use the port externally as though it were a channel

Specialized ports, sc_export



Specialized ports, sc_export

- why use **sc_export**?
  - for an IP provider, it may be desirable to export only specific channels and keep everything else private
    - **sc_export**<*T*> allows control over the interface
  - provide multiple interfaces at the top level
  - communications efficiency down the SystemC hierarchy
    - allows direct access to information (data) without intermediate channels

Specialized ports, sc_export

```
sc_export<interface> portname;
```

```
SC_MODULE(modulename) {
  sc_export<interface> portname;
  channel cinstance;
  SC_CTOR(modulename) {
    portname(cinstance);
  }
};
```

Specialized ports, sc_export

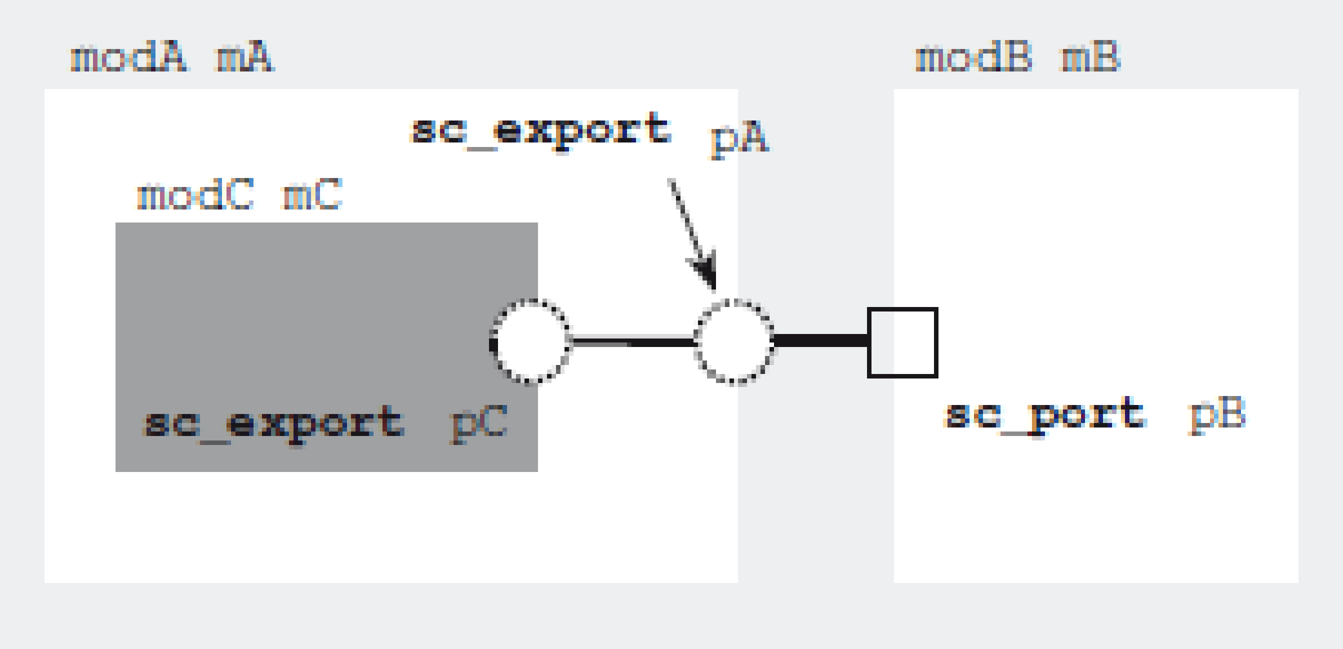
```
SC_MODULE(clock_gen) {
  sc_export<sc_signal<bool>> clock_xp;
  sc_signal<bool> oscillator;
  SC_CTOR(clock_gen) {
    SC_METHOD(clock_method);
    clock_xp(oscillator); // connect sc_signal
                          // channel
                          // to export clock_xp
    oscillator.write(false);
  }
  void clock_method() {
    oscillator.write(!oscillator.read());
    next_trigger(10,SC_NS);
  }
};
```

Specialized ports, sc_export

```
#include "clock_gen.h"
…
clock_gen clock_gen_i("clock_gen_i");
collision_detector cd_i("cd_i");
// Connect clock
cd_i.clock(clock_gen_i.clock_xp);
…
```

Specialized ports, sc_export

- **sc_export**<*T*> lets interfaces be passed up the design hierarchy



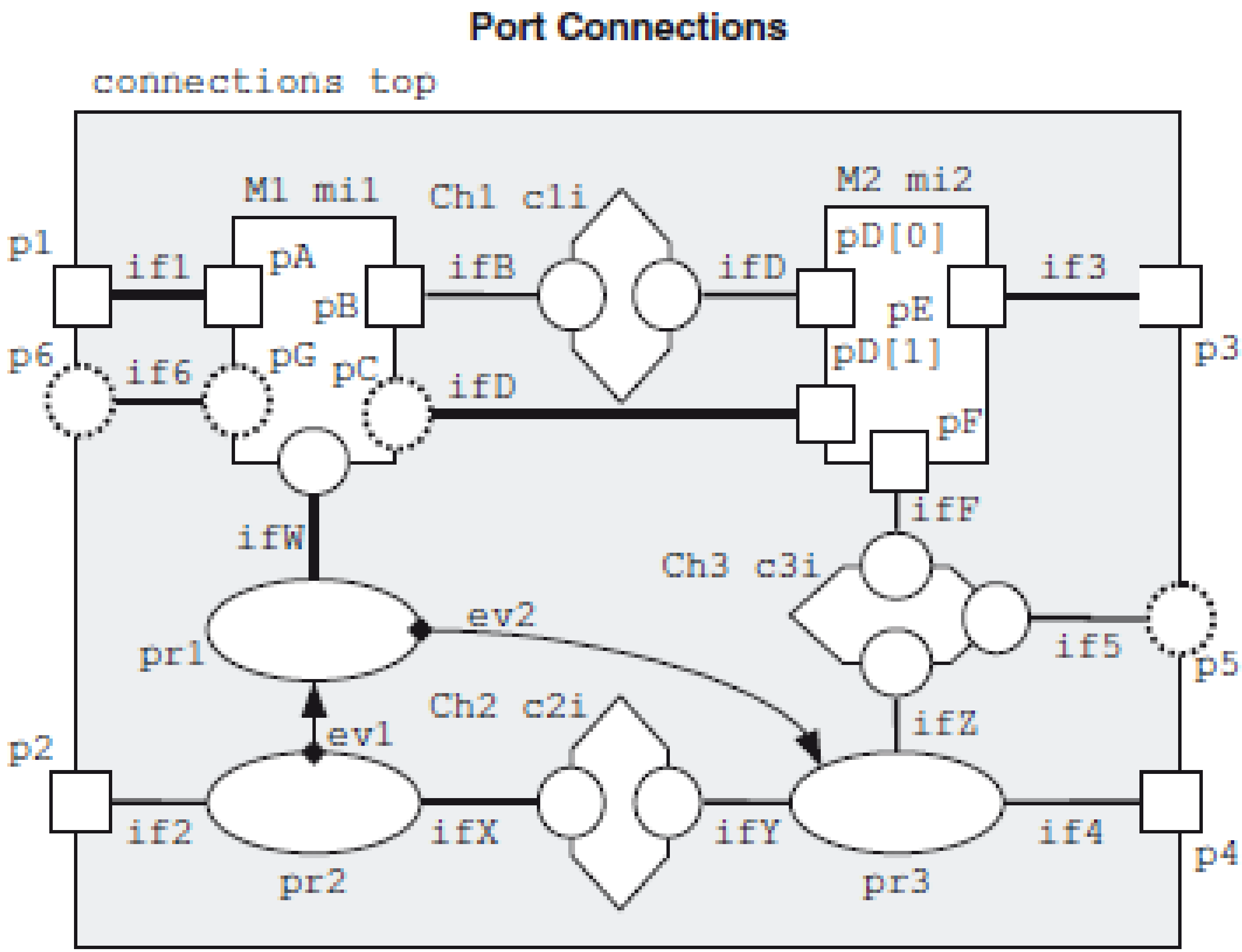
Specialized ports, sc_export

```
SC_MODULE(modulename) {
   sc_export<interface> xportname;
   module minstance;
   SC_CTOR(modulename)
   , minstance("minstance")
   {
      xportname(minstance.subxport);
   }
};
```

Specialized ports, sc_export

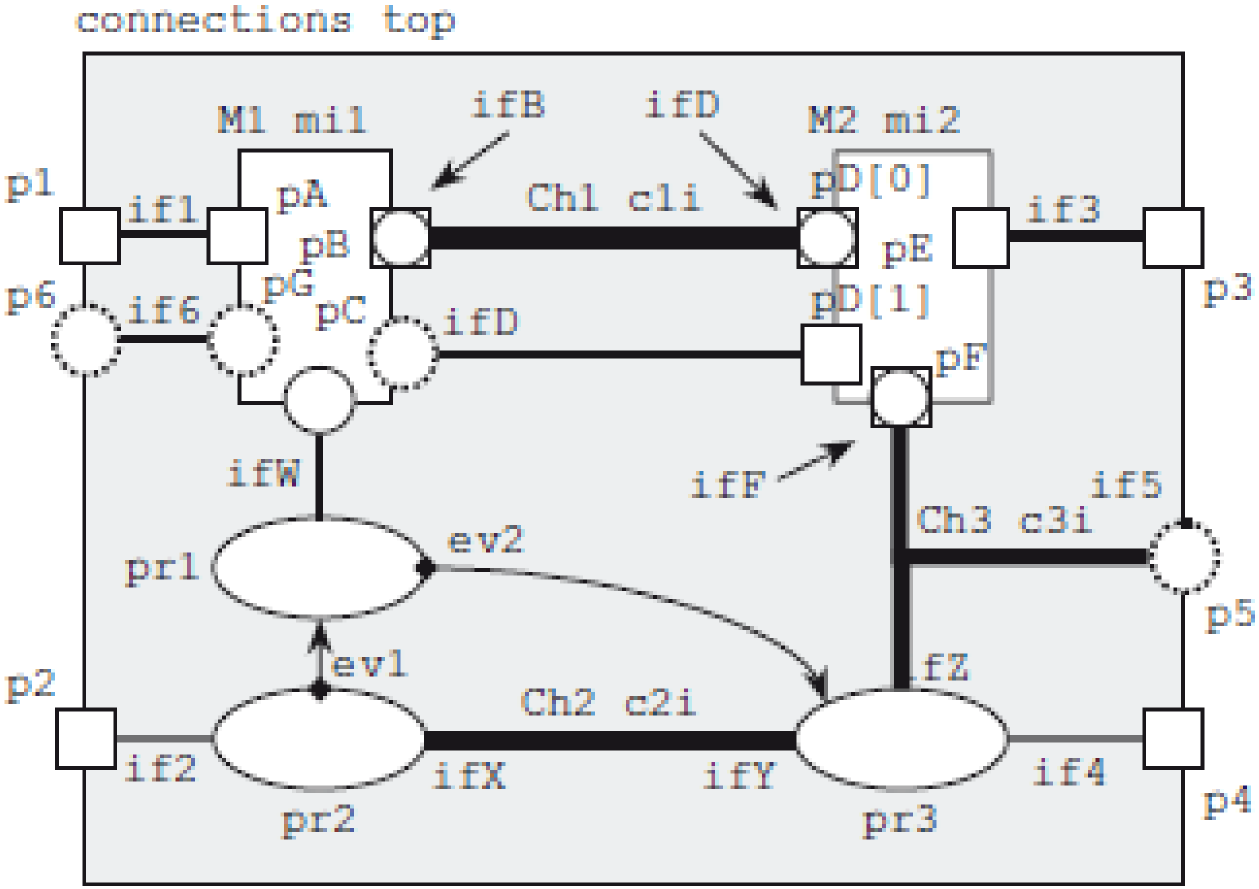
- **sc_export**<*T*> caveats:
  - it is not possible to use **sc_export**<*T*> in a static sensitivity list
  - it is not possible to have an array of **sc_export**<*T*>
  - it is possible to access the interface via the pointer operator (->)

Specialized ports, sc_export

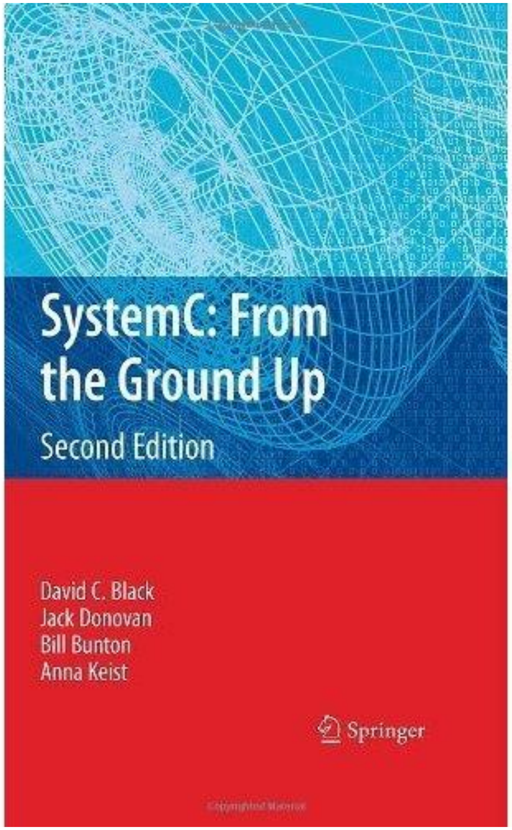


Specialized ports, sc_export

Hidden Channels



Bibliography



- David C. Black, Jack Donovan, Bill Bunton, Anna Keist, ***SystemC:From the Ground Up,*** Springer Science+Business Media, LLC 2010
  - *“*The authors designed this book primarily for the student or engineer new to SystemC. This book’s structure is best appreciated by reading sequentially from beginning to end.”