System Level Design (and Modelling for Embedded Systems)

11 – Introduction to SystemC (Part 2)

Kim Grüttner <kim.gruettner@dlr.de>
Jörg Walter <joerg.walter@offis.de>
Henning Schlender <henning.schlender@dlr.de>
Sven Mehlhop <sven.mehlhop@offis.de>

Distributed Computation and Communication, R&D Division Manufacturing
OFFIS – Institute for Information Technology

Institute of Systems Engineering for Future Mobility
German Aerospace Center (DLR-SE)

Based on the slides of M. Radetzki 2005–2008
University of Stuttgart

9. Introduction to SystemC (Part 1)

10. C++ and OOM

11. Introduction to SystemC (Part 2)

Logic data types and their operations
Data types for RT level bus modelling
Integer data types and their operations
Fixed point data types
Interfaces, Channels and Adapters

12. Transaction Level Modelling (Part 1)

13. Transaction Level Modelling (Part 2)

14. Software Refinement in SystemC

Bit Data Types

Type	bool(C++ type)	sc_logic
Values	false(0), true(1)	'0','1', 'X'(unknown) 'Z'(high-impedance)
Logic Operations	&&,||,!,etc. (see C++)	&,|,^,~
Assignment	= etc.(C++)	=, &=, |=, ^=
Comparison	==, !=	==, !=

```
Example: using namespace sc_dt;
         sc_logic a, b;
         a = '0';
         b = 'z';
         c = a | b; // result?
```

Vector Data Types

Type	sc_bv<N> vector of N bool	sc_lv<N> vector of N sc_logic
Values	e.g. "01001100"	e.g. "01XZ0011"
Logic Operations	~, &, |, ^, >>, <<	
Assignment	=, &=, |=, ^=	
Comparison	==, !=	
Selection	[int],range(int,int),(int,int)	
Concatenation	concat(vec),(vec,vec)	
Arithmetic	none	

Bit-Wise Access (Examples)

```
sc_lv<8> byte;     // vector of 8 bits, numbered 7 to 0
byte = "00000000" // setting all bits 0
byte[0] = '1';     // lhs use: writing bit #0
cout << byte[0];  // rhs use: reading bit #0
cout << byte[8];  // error
cout << byte;     // -> 00000001 (bit #0 is right)
cout << byte.range(0,7); // -> 10000000
cout << byte.range(3,0); // -> 0001
byte.range(6,4) = "111"; // lhs use: set bit #6, #5, #4
cout << byte;     // -> 01110001
sc_lv<16> halfwd; // vector of 16 bits
halfwd = (byte, byte); // concatenating 2 bytes
(byte, byte) = "1111111100000000"; // lhs use of concat
```

Signal Assignment from Multiple Processes

```
sc_signal<sc_logic> sig;
int                 var;
```

```
SC_THREAD(driver1);
void driver1()
{
  sig.write('1');
  var = 1;
}
```

```
SC_THREAD(driver2);
void driver2()
{
  sig.write('0');
  var = 0;
}
```

- The order of evaluating processes that are runnable in the same delta cycle is not defined in SystemC (can driver1 first or driver2 first).
- The resulting value in sig, var depends on the evaluation order and is therefore non-deterministic.

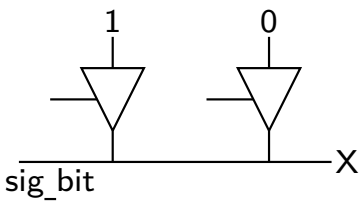
Resolved Signals for Bus Modelling

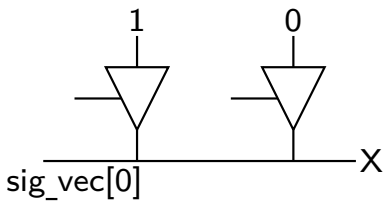
```
sc_signal_resolved sig_bit;
sc_signal_rv<32>   sig_vec;
```

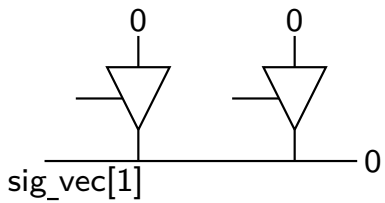
```
0
0...0X
```

```
SC_THREAD(driver1);
void driver1()
{
  sig_bit.write('1');
  sig_vec.write(1);
}
```

```
SC_THREAD(driver2);
void driver2()
{
  sig_bit.write('0');
  sig_vec.write(0);
}
```







Resolution Table

- Resolution is performed **bit by bit**
- for each bit, resulting value is determined by **table lookup**
- More than 2 drivers
  → **repeated table lookup** with intermediate result

	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

Example (2 drivers):
sig_bit = '1' || sig_bit = '0'
⇒ resulting value 'X'

Example (3 drivers):
sig_bit = '1' || sig_bit = 'Z' || sig_bit = 'X'
⇒ intermediate value '1'
⇒ resulting value 'X'

Arithmetic Data Types

Type	sc_int<N> vector of sign and $N-1 \leq 63$ binary digits	sc_uint<N> vector of $N \leq 64$ binary digits
Values	signed 2's compl.	unsigned
Logic Operations	same as logic data types	
Arithmetic Ops	+, -, *, /, %, ++, --	
Assignment	=, &=, |=, ^=, +=, -=, *=, /=, %=	
Comparison	==, !=, >, <, <=, >=	
Selection, Concat	same as logic data types	

Note: can mix sc_int, sc_uint with C++ integer data types

e.g.: sc_int + int is possible

Arbitrary Precision Arithmetics

Type	sc_**big**int<N> vector of sign and $N-1$ binary digits	sc_**big**uint<N> vector of N binary digits
Values	signed 2's compl.	unsigned
Operations	same as sc_int, sc_uint	

Notes:

- can mix sc_bigint, sc_biguint with sc_int, sc_uint and C++ integer data types
- **precision is 64 if no big data type** is involved in an expression
- precision is **arbitrary if big data type involved**

Literals

- **Values of SystemC vector types can be written as:**

```
sc_lv<8> byte;
```

  - bitstrings

```
byte = "10101010";
byte = "1010XXXX";
```

  - binary string

```
byte = "0b10101010";
```

  - decimal string

```
byte = "0d170";
```

  - hex string

```
byte = "0xAA"; // what if "0X11"?
```

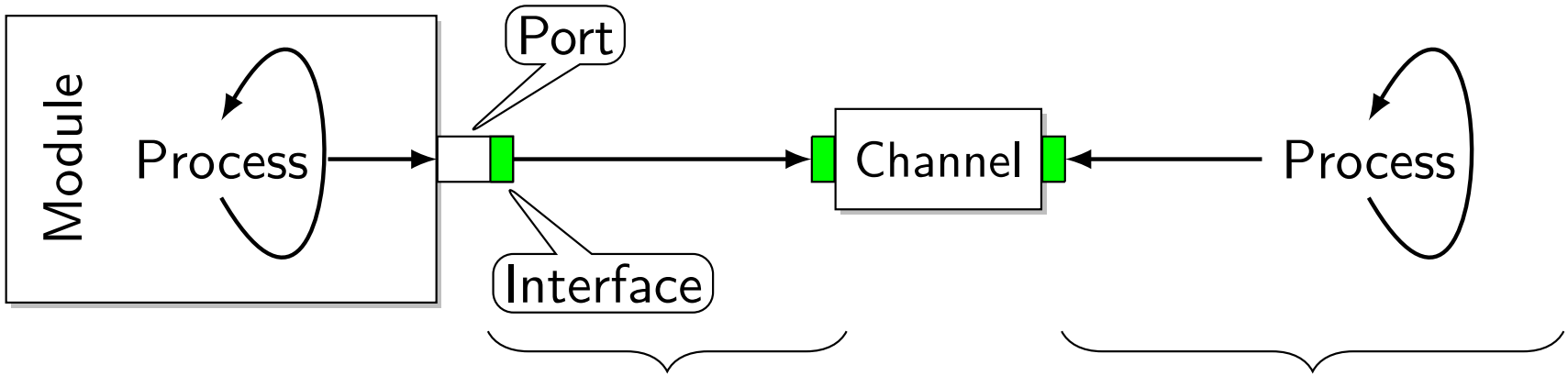
  - C++ number

```
byte = 170;
byte = 0xAA;
```

Fixed Point Data Types

Type	`sc_fixed<..>`	`sc_ufixed<..>`
Values	signed 2's compl.	unsigned
Parameters	wl: total number of bits (word length) iwl: bits before .(integer word length) q_mode: quantization mode o_mode: overflow mode, e.g. saturated n_bits: (related to saturation)	
Arithmetic Ops	+, -, *, /, >>, <<, ++, –	
Assignment	=, +=, -=, *=, /=	
Comparison	==, !=, >, <, <=, >=	
Selection, Concat	same as logic data types	

SystemC Communication Concept



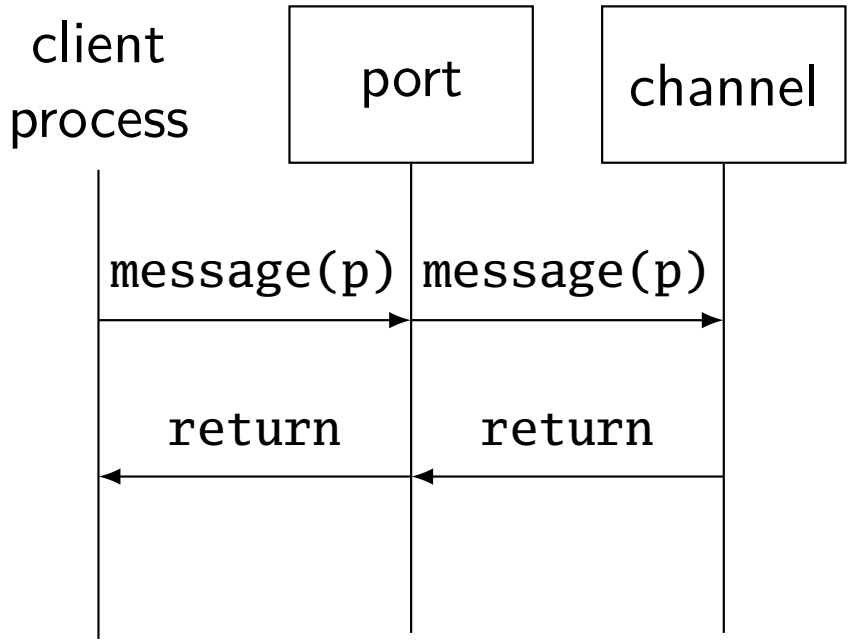
	channel access via port	direct access to channel
Interface		
Message	port forwards messages	process passes
Call	from the process	messaged directly
	to the channel	to the channel

```
port->message(parameters)
instance.port(channel)
```

```
channel.message(params)
```

SystemC Communication Concept

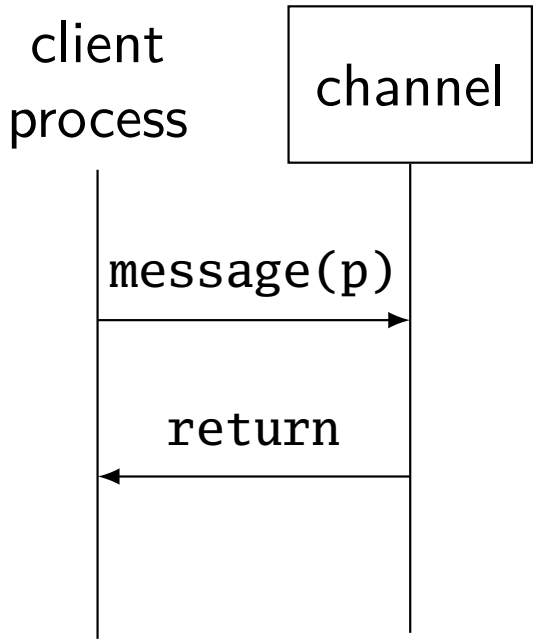
Interface Method Call (indirect)




```
port->message(parameters)
instance.port(channel)
```

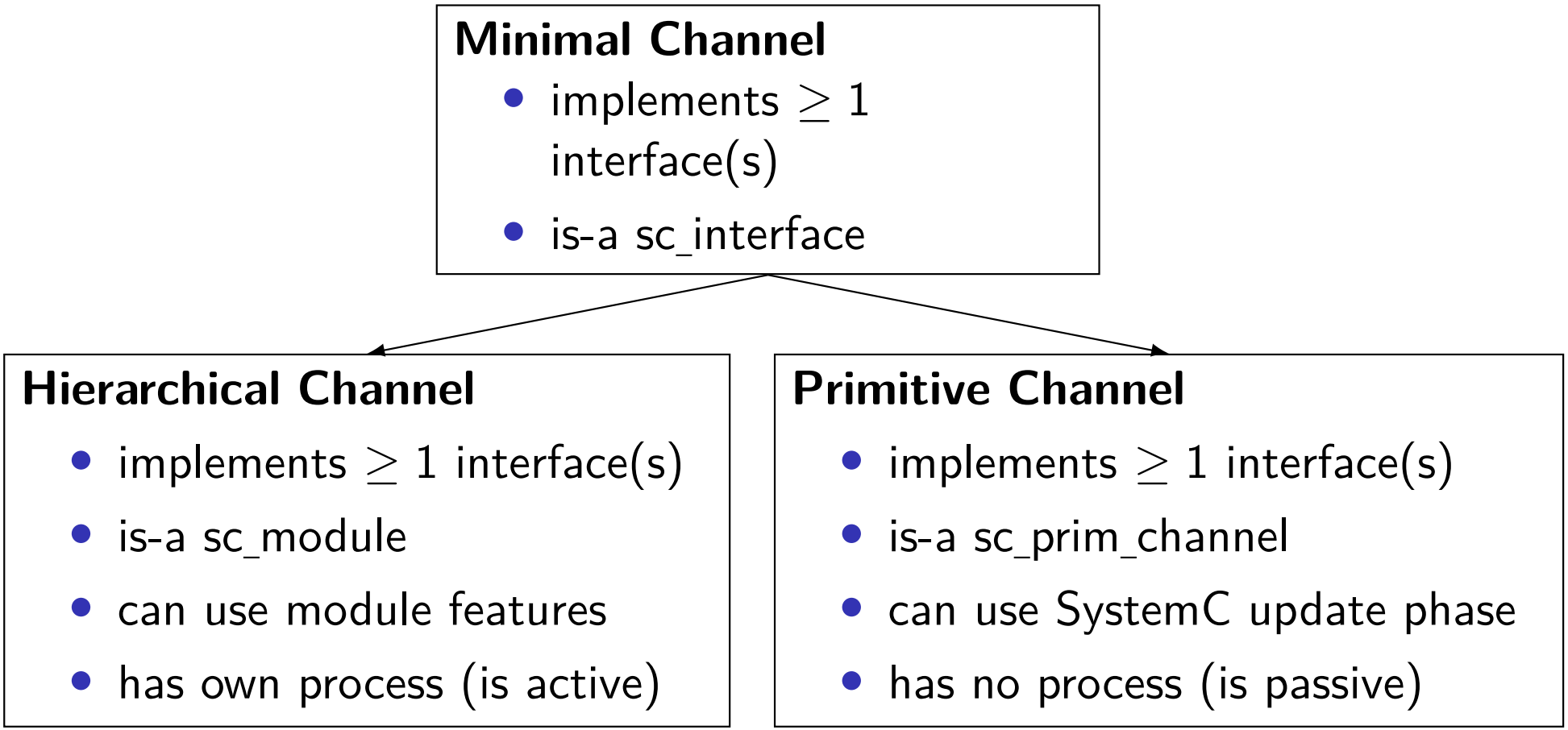
binding

Direct call to channel




```
channel.message(params)
```

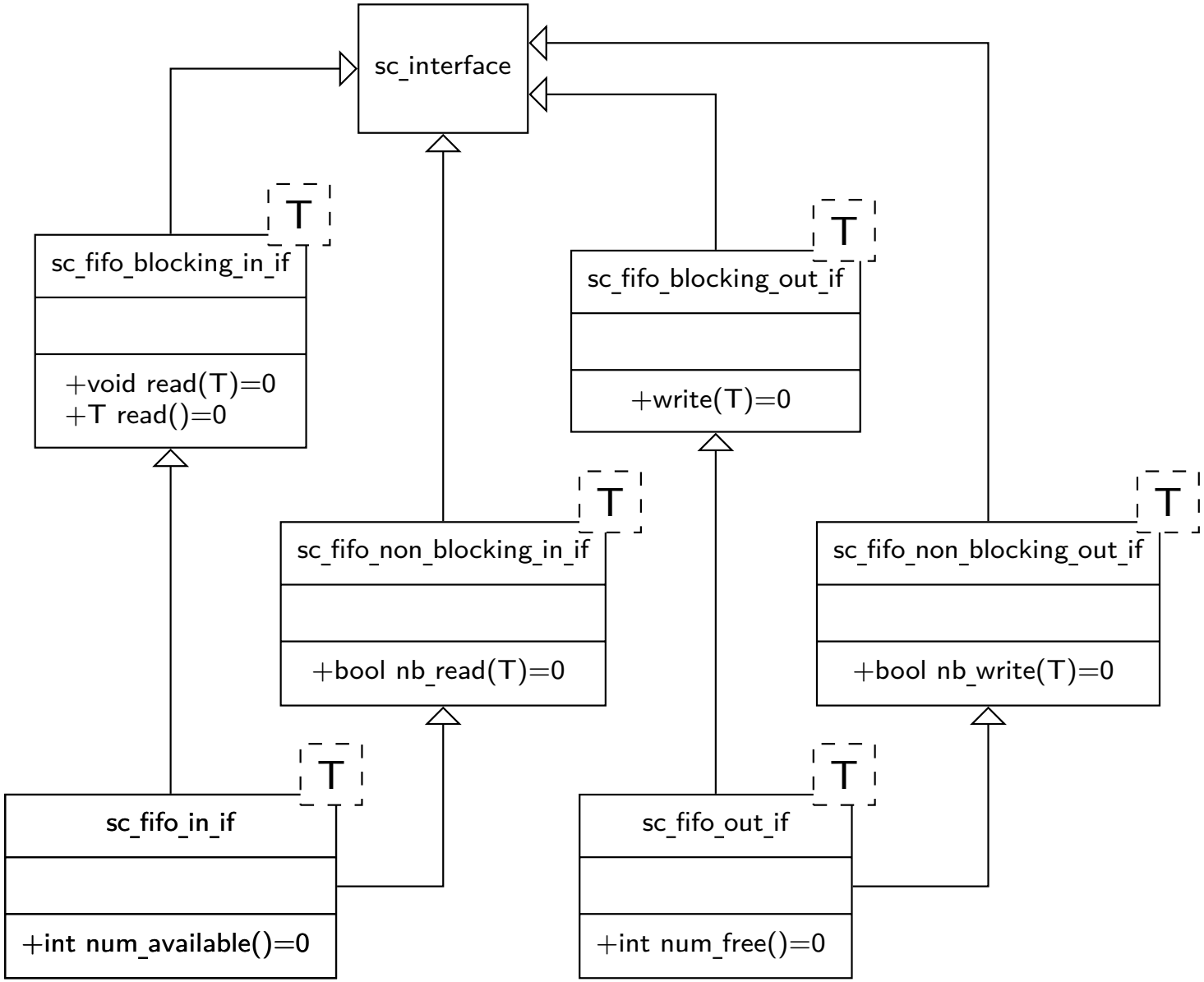
Classification of Channels



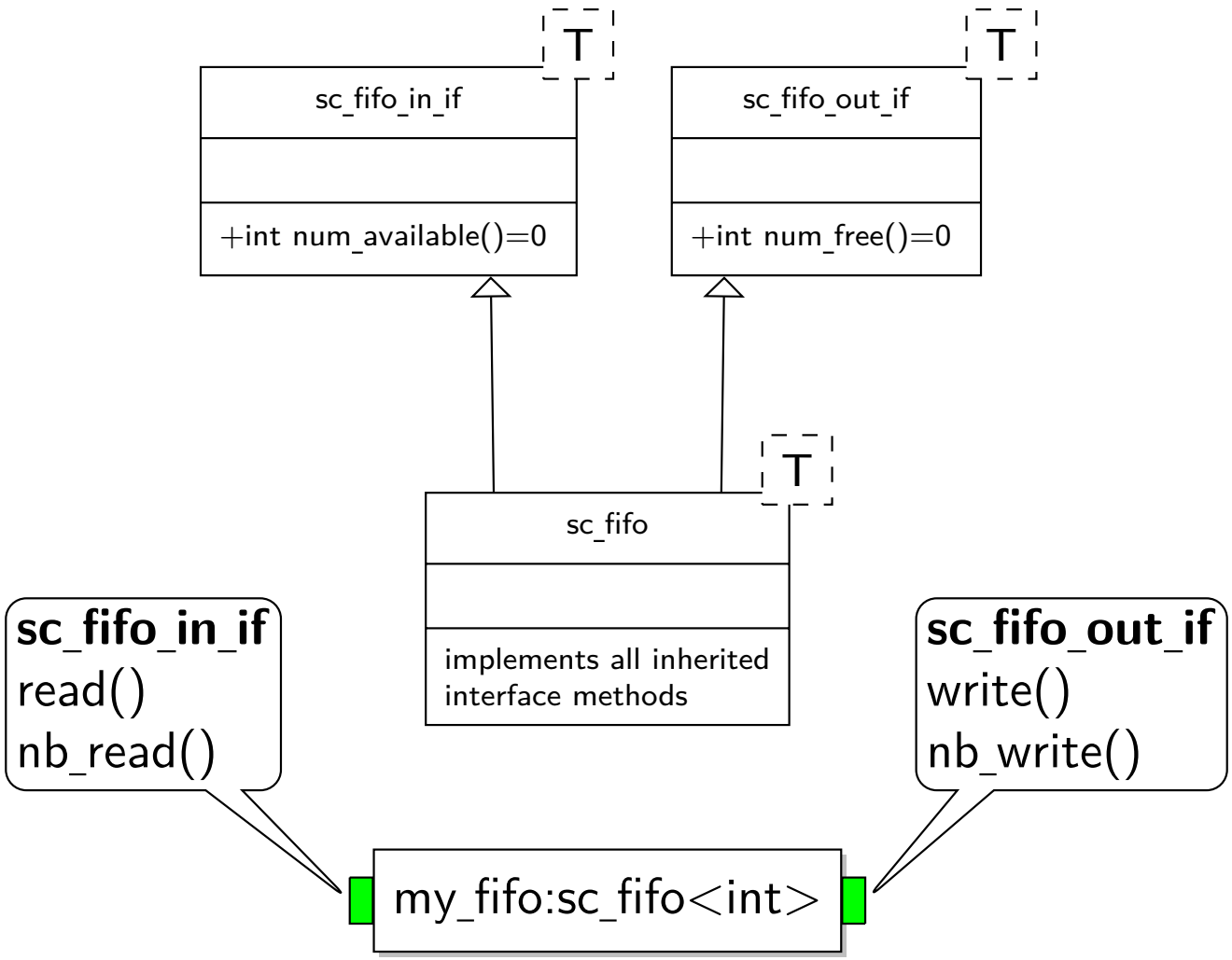
User-defined channels are often hierarchical channels

SystemC built-in channels are primitive channels

SystemC Interfaces (e.g.:FIFO)



SystemC Interfaces (e.g.:FIFO)



C++ Code Excerpts

File: sc_fifo_ifs.h

```
template<class T>
class sc_fifo_nonblocking_in_if
:virtual public sc_interface
{
  ...
};
```

avoids inheritance of multiple copies of things from sc_interface

File: sc_fifo.h

```
template<class T>
class sc_fifo
:public sc_fifo_in_if<T>
,public sc_fifo_out_if<T>
,public sc_prim_channel
{
  ...
};
```

sc_fifo is-a primitive channel

SystemC Ports

General port declaration

```
sc_port<my_interface,2>my_port;
```

File: sc_port.h

```
template<class IF, int N = 1>
class sc_port
...
```

Maximum number of channels that can be connected to the port (default here: 1)

Interface of the channel to which the port shall be bound

using pre-defined specialized port

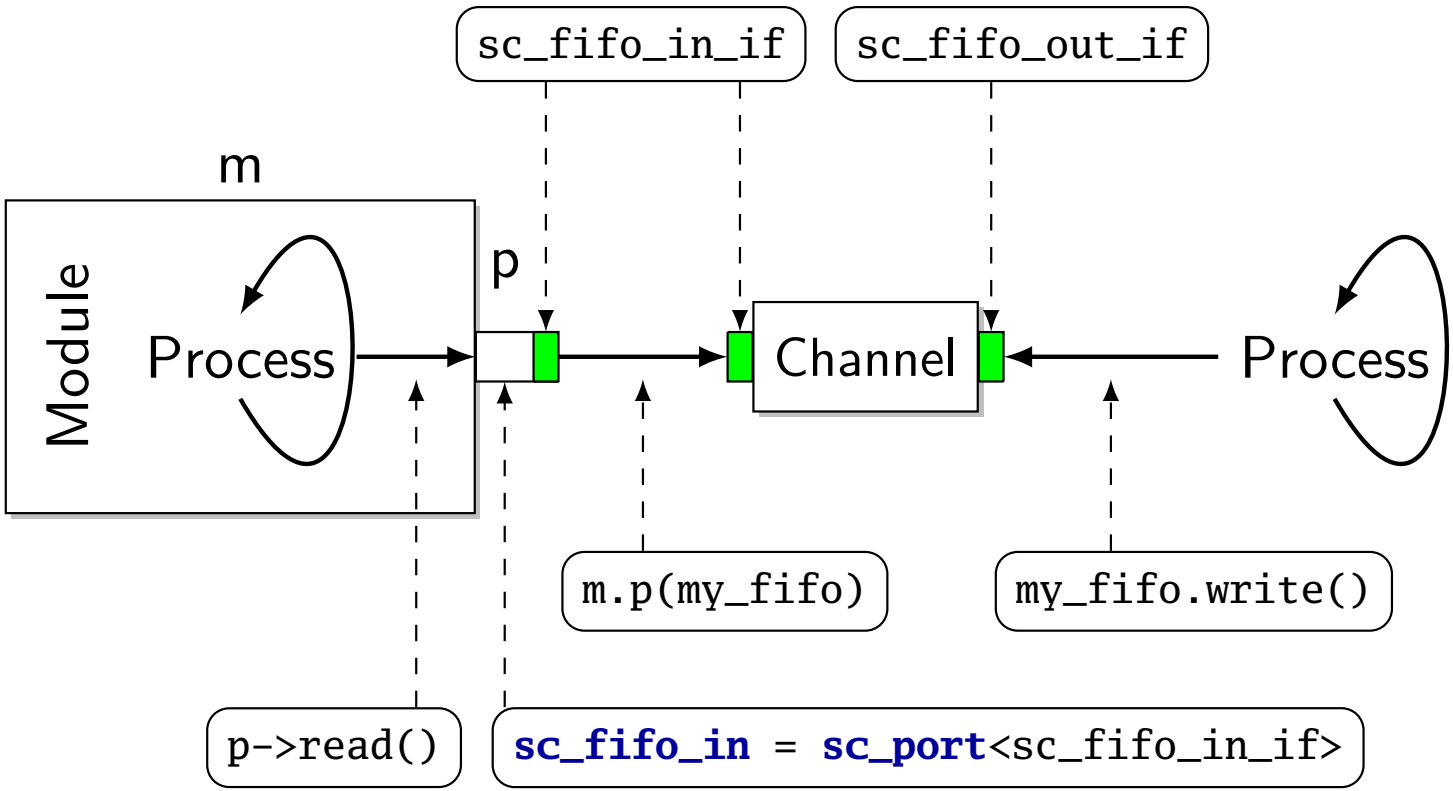
```
sc_fifo_in<int> my_port;
```

File: sc_fifo_ports.h

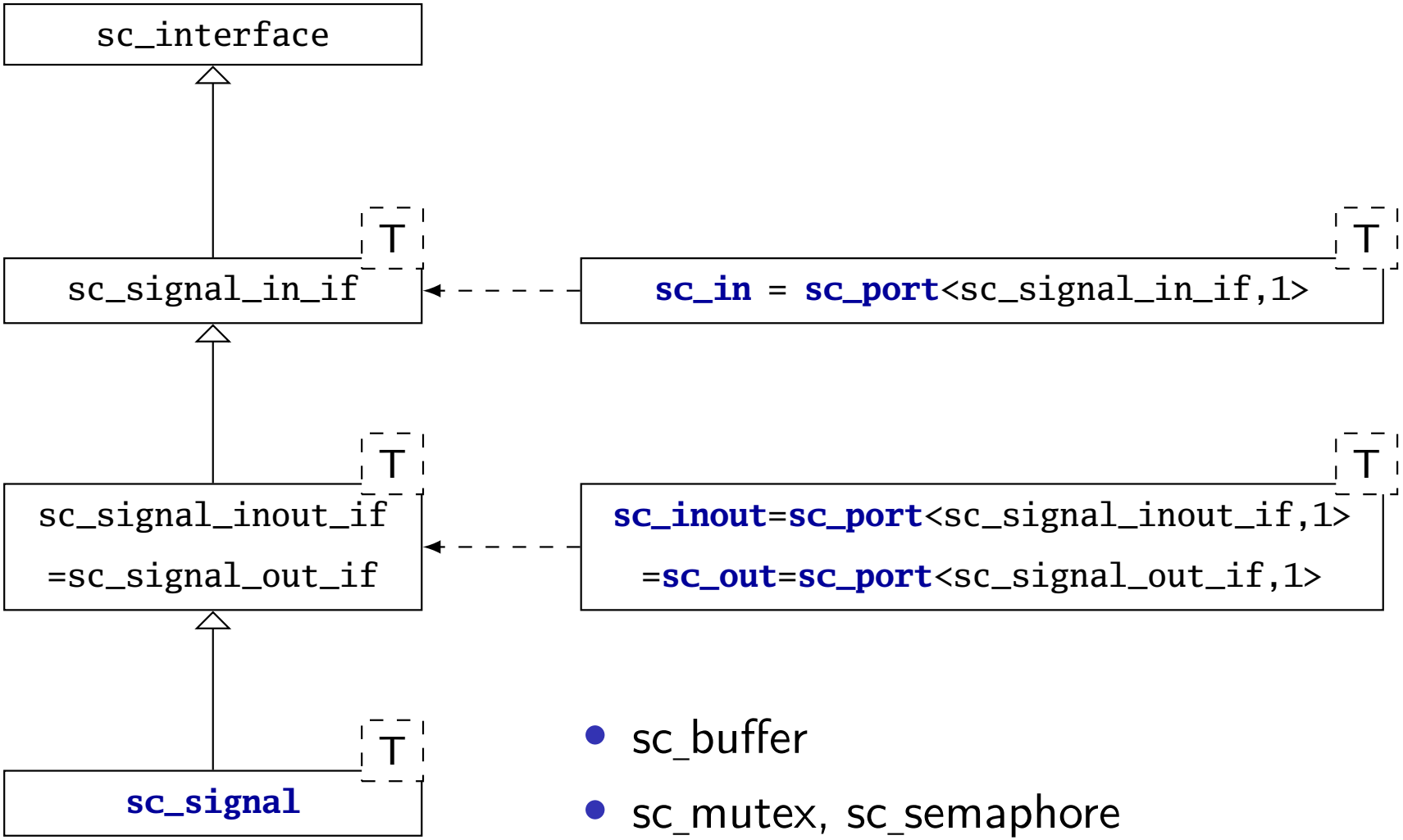
```
template<class T>
class sc_fifo_in
:public
    sc_port<sc_fifo_in_if<T>,0>
    {...};

template<class T>
class sc_fifo_out
:public
    sc_port<sc_fifo_out_if<T>,0>
    {...};
```

Putting It All Together

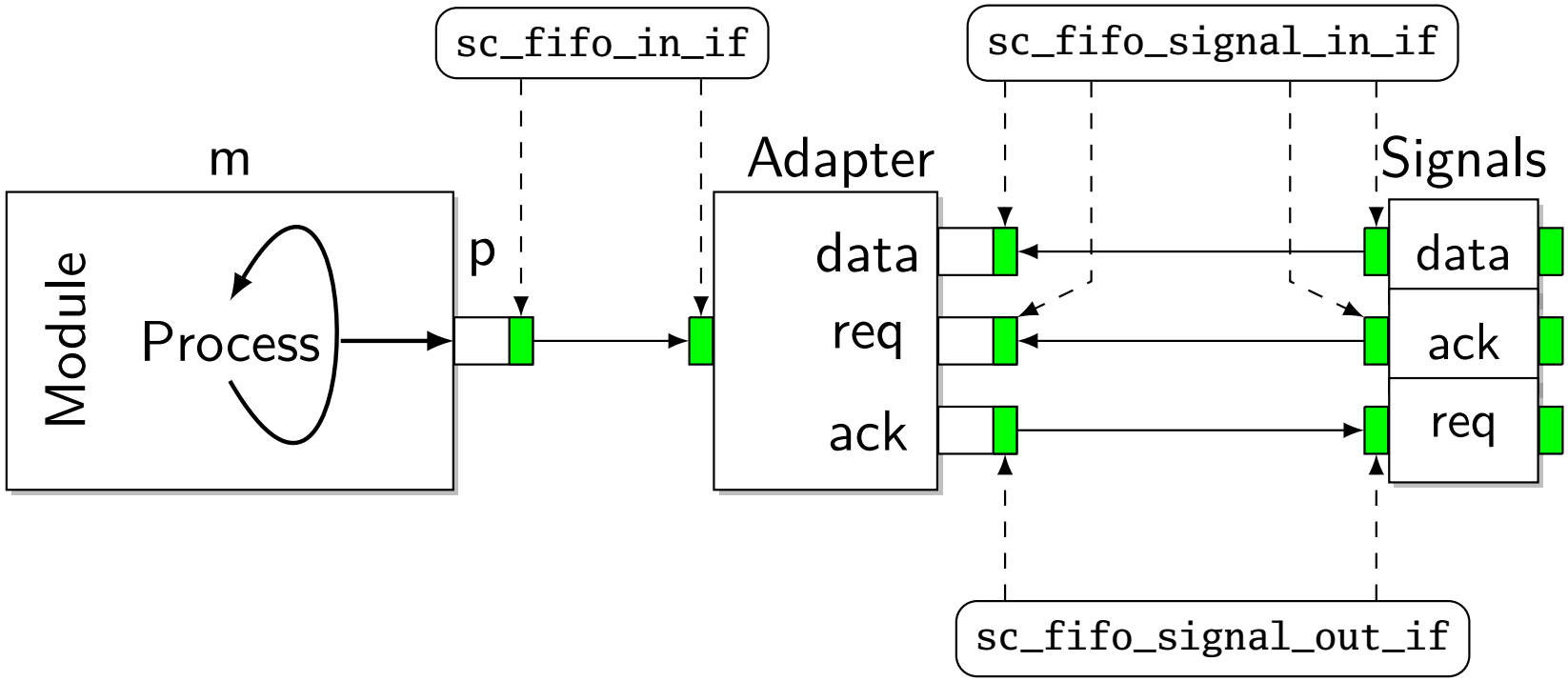


Other SystemC Channels / Ports



- sc_buffer
- sc_mutex, sc_semaphore
- your own IFs / channels / ports

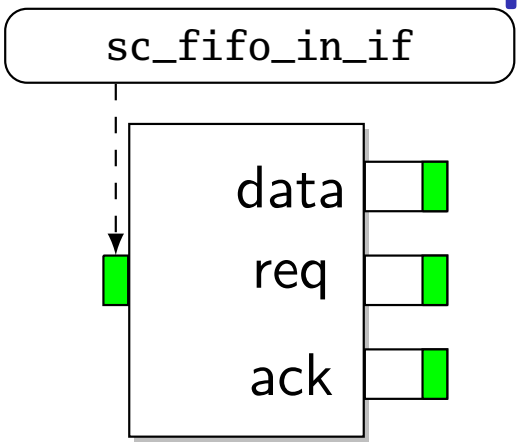
Interface Adapters



An **adapter** enables connection between **different interfaces**

An adapter that translates between a **signal interface** and a more **abstract interface** is called a **transactor**.

Interface Adapters



The adapter

- implements an interface to connect to a port
- has port(s) to connect to the other interface

```
template<class T>
class Adapter
: public sc_module
, public sc_fifo_in_if<T>
{
public:
  // implement sc_fifo_in_if
  T read();
  void read(T&);
  bool nb_read(T&);
  int num_available();
  sc_event& data_written_event();
  // ports to connect to other side
  sc_in<T>     data;
  sc_in<bool>  ack;
  sc_out<bool> req;
};
```

Adapter is a **hierarchical channel**

Method Implementation

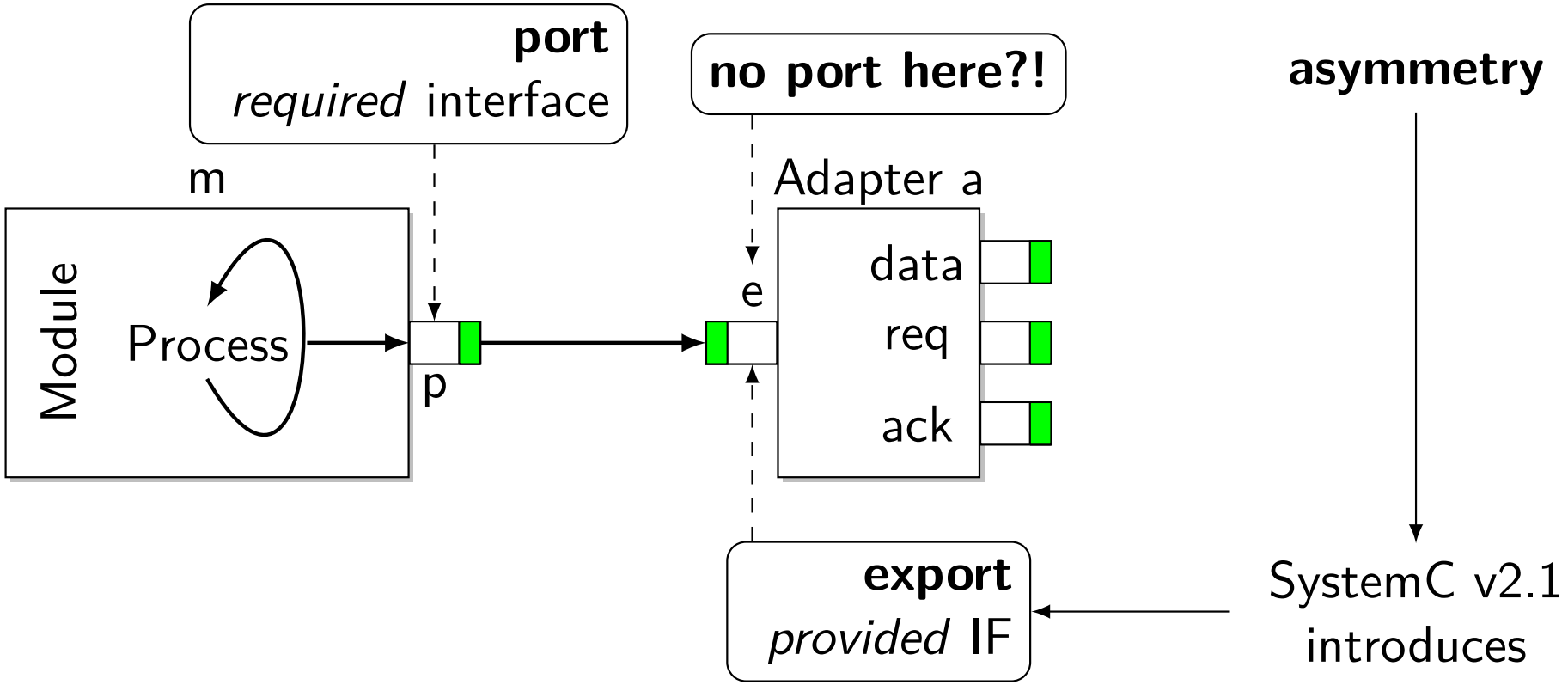
```
template<class T>
T Adapter<T>::read()
{
  req.write(true);
  wait( ack.posedge_event() );
  req.write(false);
  return data.read();
}
```

translate call to `sc_fifo_if::read()` into call(s) of the interface on the other side

```
template<class T>
int Adapter<T>::num_available()
{
  return 0;
}
```

dummy implementation of `sc_fifo_if` methods that are not needed or that cannot be meaningfully implemented

Exports

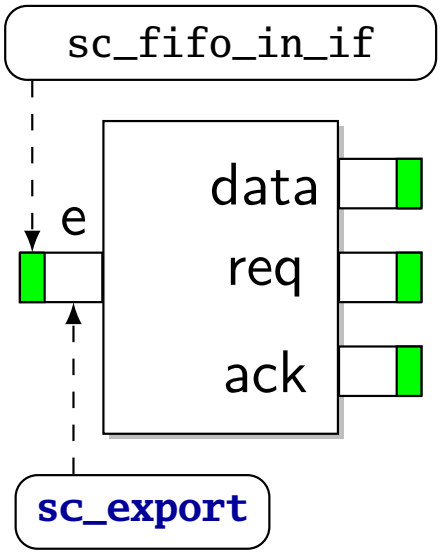


connection/binding:
`m.p( a.e )`

alternative syntax:
`m.p.bind( a.e )`

- Note: `sc_modules` other than adapters may have exports, too.

Exporting an Interface



in constructor:

e(*this)

or:

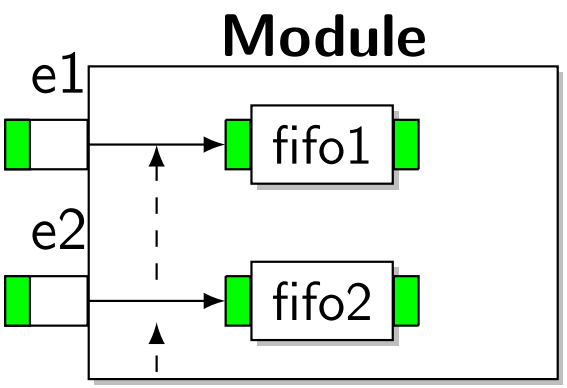
e.bind(*this)

```
template<class T>
class Adapter
: public sc_module
, public sc_fifo_in_if<T>
{
public:
  // implement sc_fifo_in_if
  T read();
  ...
  // export the interface
  sc_export<sc_fifo_in_if<T> > e;
  // ports to connect to other side
  sc_in<T>     data;
  sc_in<bool>  ack;
  sc_out<bool> req;
};
```

Adapter is a **hierarchical channel**

need white-space here

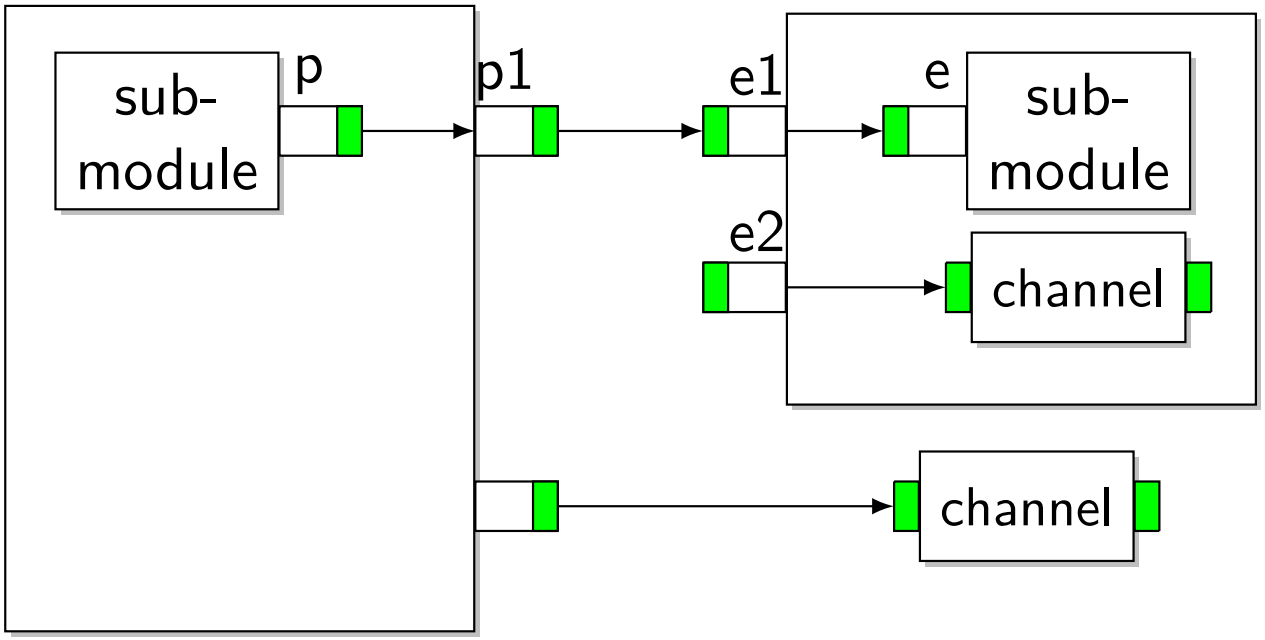
Exporting Internal Channels




```
template<class T>
class Module
: public sc_module
{
public:
  sc_fifo fifo1;
  sc_fifo fifo2;
  // export the interface
  sc_export<sc_fifo_in_if<T> > e1;
  sc_export<sc_fifo_in_if<T> > e2;
  SC_CTOR(Module)
  {
    e1( fifo1 );
    e2( fifo2 );
  }
};
```

sc_module can have only one interface of the same kind, but **multiple exports of the same interface**

Hierarchical Connections



$a \rightarrow b$
Binding: `a.bind(b)`

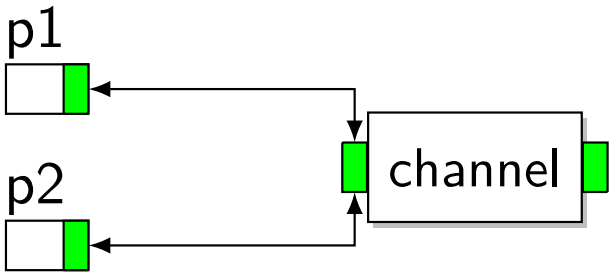
Port

Export

- **Ports can be bound to**
  - ports at higher hierarchy level
  - channels
  - exports at same hierarchy level
- **Exports can be bound to**
  - exports at lower hierarchy level
  - channels

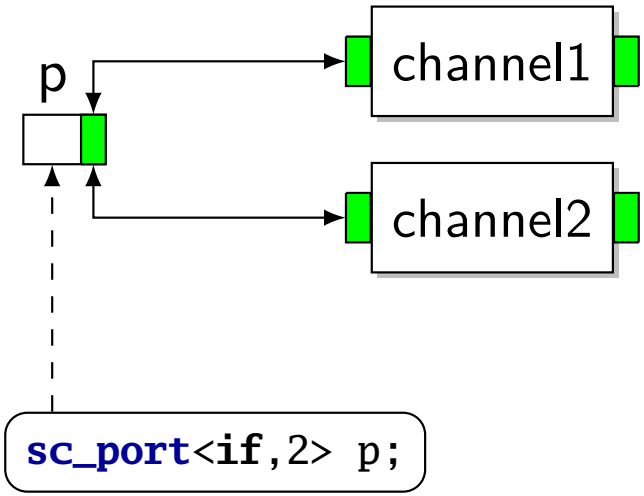
m:1, 1:n Connections

- up to now: 1:1 (1 port with 1 channel or 1 export)
- m:1 - m ports connected to same channel / export



channel must take care of access arbitration

- 1:n - n channels / exports connected to same *multi*-port



multiport is like an array
`p[0]` - access to one connected channel
`p[1]` - access to other connected channels
`p.size()` - number of connected channels