

# Introduction à SystemC

Hiérarchie et modules

Tarik Graba

P4 2015-2016



Télécom ParisTech

# Table des matières

<b>1 Les modules</b>	<b>3</b>
sc_module . . . . .	3
Le constructeur d'un module . . . . .	4
Les macros prédéfinies . . . . .	4
Constructeurs additionnels . . . . .	5
<b>2 Les ports</b>	<b>6</b>
sc_in, sc_out, sc_inout . . . . .	6
Comment utiliser les ports . . . . .	7
Types résolus . . . . .	8
<b>3 Instances</b>	<b>9</b>
Dans le sc_main . . . . .	9
Dans un module . . . . .	10

# 1 Les modules

## **sc\_module**

Les modules sont les éléments principaux pour construire une description hiérarchique en SystemC.

Un module SystemC est une classe qui hérite de la classe `sc_module`

```
struct mon_module : sc_module {  
    ...  
};
```

Un module ne peut être déclaré après le début de la simulation.

---

Un module SystemC est une classe C++ dans laquelle tous les champs sont publics par défaut.

Il peut donc avoir des attributs et des méthodes comme toute classe C++. Les attributs peuvent bien sûr être tout objet C++ ou SystemC, même un autre module.

De plus, on peut séparer la déclaration de la classe de la définition des méthodes dans deux fichiers (.h et .cpp).

### **rappel**

En C++ :

```
struct X {  
    ...  
};
```

est équivalent à :

```
class X {  
public:  
    ...  
};
```

---

## Le constructeur d'un module

Un module doit avoir un nom à l'instanciation. Ce nom est passé comme argument de son constructeur et doit être transmis à la classe parent `sc_module`.

Par exemple :

```
mon_module(int i, sc_module_name n):sc_module(n){ ... }
```

La méthode `name()` permet de récupérer ce nom durant l'exécution de la simulation.

---

La classe `sc_module_name` est une classe intermédiaire qui est utilisée pour définir un nom de module à partir d'une chaîne de caractères. À l'usage elle est transparente.

---

## Les macros prédéfinies

Deux macros sont définies pour simplifier cette déclaration :

- `SC_MODULE` pour déclarer le module
  - `SC_CTOR` pour définir le constructeur
- 

`SC_MODULE(mon_module)` est équivalent à `struct mon_module : sc_module`.

`SC_CTOR(mon_module)` est équivalent à `mon_module(sc_module_name n) : sc_module(n)`

On peut donc déclarer un module en écrivant :

```
SC_MODULE(mon_module)
{
    // déclaration de méthodes et attributs

    SC_CTOR(mon_module)
    {
        // Initialisation
    }
};
```

**Remarque :** On verra par la suite que `SC_CTOR` ajoute une information supplémentaire.

---

## Constructeurs additionnels

Si un constructeur additionnel est ajouté il doit aussi appeler le constructeur de la classe parent `sc_module`

Par exemple :

```
mon_module(int i, sc_module_name n):sc_module(n)
{
    ...
}
```

Dans ce cas, la macro `SC_CTOR` ne peut plus être utilisée.

## 2 Les ports

### sc\_in, sc\_out, sc\_inout

SystemC définit trois types de ports sous la forme de templates :

- `sc_in<T>` les entrées de type T
- `sc_out<T>` les sorties de type T
- `sc_inout<T>` les ports bidirectionnels de type T

---

Comme pour les signaux, le template T peut être tout SystemC ou C++. De plus, la surcharge de la fonction `sc_trace` pour ce type est obligatoire.

À la construction, on peut donner un nom au port. Ce nom, est utile pour identifier certains problèmes à la simulation.

Si un nom n'est pas donné, un nom générique sera créé. Dans tous les cas, ce nom sera hiérarchique et dépendra du nom du module.

Exemple :

```
#include <systemc.h>
```

```
SC_MODULE(foo)
{
    sc_in<bool> clk;
    sc_in<bool> rst;
    sc_out<sc_uint<8> > data;

    SC_CTOR(foo):clk("clk"), data("data")
    {
        cout
            << "module : " << name() << endl
            << "clk    : " << clk.name() << endl
            << "rst    : " << rst.name() << endl
            << "data   : " << data.name() << endl
            ;
    }
};
```

```

int sc_main(int argc, char * argv[])
{
    foo bar("foobar");
    return 0;
}

```

Les trois types de ports héritent d'une classe primaire `sc_port`. Cette classe primaire permet d'étendre le concept de port.

## Comment utiliser les ports

Les ports SystemC doivent être connectés à des signaux avant de pouvoir les utiliser. Lire ou écrire dans un port non connecté entraîne une erreur durant la simulation.

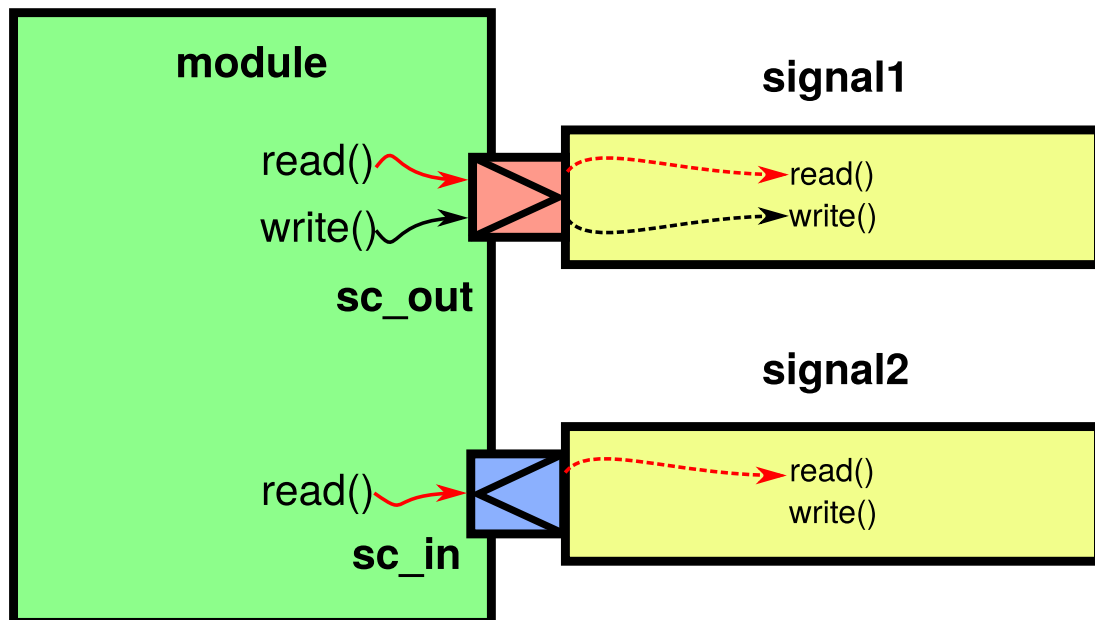


FIG. 2.1: Ports et signaux

En fonction du types de ports, ils implémentent les méthodes :

- `read()`
- `write()`

Les opérateurs d'affectation sont surchargés pour que l'appel à ces méthodes soit transparent.

Ces méthodes ne font rien à part appeler les méthodes équivalentes des signaux connectés aux ports. Ce sont en réalité des coquilles vides. Quand on lit ou qu'on écrit dans un port, c'est en réalité au signal qui lui est connecté qu'on accède.

On est parfois amené à appeler explicitement les méthodes `read` et `write` particulièrement quand les transtipages automatiques fonctionnent pas.

---

## Types résolus

Comme pour les signaux, il existe des ports pour les types résolus :

- `sc_in_resolved`, `sc_out_resolved` et `sc_inout_resolved`
- `sc_in_rv<N>`, `sc_out_rv<N>` et `sc_inout_rv<N>`

Il doivent être connectés à des signaux du bon type.



## 3 Instances

### Dans le `sc_main`

Les modules peuvent être instanciés dans la fonction `sc_main`.

Ils peuvent être connectés à des signaux déclarés au même niveau.

L'opérateur `()` est surchargé pour connecter les ports d'un module à des signaux. En interne, la méthode `bind` est appelée et des vérifications sont faites avant le début de la simulation.

---

#### Exemple :

```
#include <systemc.h>

SC_MODULE(foo) {
    sc_in<bool> i;
    SC_CTOR(foo):i("i") {}
};

SC_MODULE(bar) {
    sc_out<bool> o;
    SC_CTOR(bar):o("o") {}
};

int sc_main(int argc, char * argv[])
{
    sc_signal<bool> s("s");

    foo foo_i("foo");
    bar bar_i("bar");

    // connexion
    foo_i.i(s);
    bar_i.o(s);

    return 0;
}
```

---

## Dans un module

Les sous modules sont déclarés comme des attributs du modules.

Les connexions se font alors dans le constructeur du module.

Un port peut être connecté soit :

- un autre port
- un signal interne

---

**Exemple :**

```
#include <systemc.h>
```

```
SC_MODULE(foo) {  
    sc_in <bool> i;  
    sc_out<bool> o;  
    SC_CTOR(foo):i("i"),o("o") { }  
};
```

```
SC_MODULE(bar) {  
    sc_in <bool> i;  
    sc_out<bool> o;  
    SC_CTOR(bar):i("i"),o("o") { }  
};
```

```
SC_MODULE(foofoo) {  
    // entrée/sortie  
    sc_in <bool> i;  
    sc_out<bool> o;  
    // interne  
    sc_signal<bool> s;  
    // sous modules  
    foo foo_i;  
    bar bar_i;  
  
    SC_CTOR(foofoo)  
        :i("i"),o("o"),s("s"),  
        foo_i("foo"), bar_i("bar")
```

```

{
    // connexions aux I/O
    foo_i.i(i);
    bar_i.o(o);
    // connexion interne
    foo_i.o(s);
    bar_i.i(s);
}
};

int sc_main(int argc, char * argv[])
{
    sc_signal<bool> i("i");
    sc_signal<bool> o("o");

    foobar uut("foobar");

    uut.i(i);
    uut.o(o);

    return 0;
}

```

---