Introduction à SystemC

Hiérarchie et modules

Tarik Graba

2020-2021

Table des matières

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 sur 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(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 chaine 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 déclarant 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.

Dans le cas où le constructeur n'est pas déclaré avec `SC_CTOR`, alors, si le module contient des processus, on **doit** utiliser la macro `SC_HAS_PROCESS`.

Exemple d'utilisation de `SC_HAS_PROCESS`

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

    mon_module(int i, sc_module_name n):sc_module(n)
   {
   // Initialisation
   }

   SC_HAS_PROCESS(mon_module);
};
```

La déclaration des processus sera vue dans la suite du cours.

---

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 tmplate T peut être tout SystemC ou C++. De plus, la surcharge de la fonction `sc_trace` pour ce type est obligatoire.

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

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;
}
```

Ou en utilisant les constructions de C++11 :

```
#include <systemc.h>

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

    SC_CTOR(foo)
    {
        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;
}
```

Voir aussi la macro SC_NAMED introduite avec SystemC 2.3.3.

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 entraine une erreur durant la simulation.

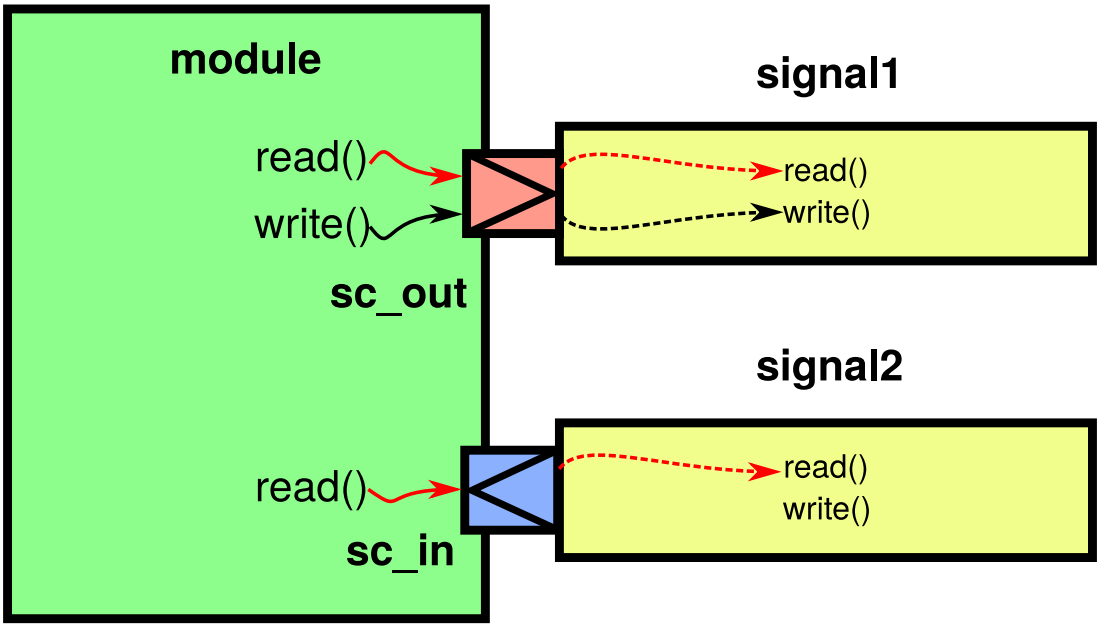

FIG. 2.1: Ports et signaux

En fonction du type 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 vide. Quand on lit ou qu'on écrit dans un port, c'est en réalité au signal qui lui est connecté qu'on accède.

Donc, lire ou écrire sur un port non connecté ne peut pas fonctionner. Pour éviter cela, durant la phase d'élaboration, avant le début de la simulation, des vérifications sont faites pour garantir que tous les ports sont connectés (pour s'en rendre compte, il faut appeler `sc_start`).

On est parfois amené à appeler explicitement les méthodes `read` et `write` particulièrement quand les transtipages automatiques ne 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>`

Ils 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 {"i"};
   SC_CTOR(foo) {}
};

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

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

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

   //foo_i.i(bar_i.o);// NON!
   // connexion à travers le signal s
   foo_i.i(s);
   bar_i.o(s);

   return 0;
```

```
}
```

Dans un module

Les sous modules sont déclarés comme des attributs du module.

Les connexions se font alors dans le constructeur du module.

Un port peut être connecté soit à :

- un autre port
- un signal interne

Exemple :

```
#include <systemc.h>

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

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

SC_MODULE(foobar)
{
   // entrée/sortie
   sc_in <bool> i {"i"};
   sc_out<bool> o {"o"};
   // interne
   sc_signal<bool> s;
   // sous modules
   foo foo_i {"foo_i"};
   bar bar_i {"bar_i"};

   SC_CTOR(foobar)
   {
```

```
      // 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;
}
```

Travail à faire

Ajoutez au `sc_main` le nécessaire pour relier l'entrée du module `foobar` à un signal d'horloge périodique. Tracez ensuite l'état de tous les signaux (internes et externes).

---

4 Les exports

sc_export

SystemC 2.2 ajoute les export qui permettent d'accéder à un signal interne à partir de l'extérieur d'un module.

Sûr et transparent

On peut connecter un export directement à un port.

On a la garantie à la compilation qu'on propagera les même types.

Un sc_export prend en argument de template, le type de signal auquel il permettra d'accéder.

Pour appeler directement les méthodes du signal (read, write) à travers un sc_export, on doit utiliser l'opérateur ->.

Un sc_export peut aussi être connecté à un sc_port. Le port est alors connecté (bind) au signal exporté. La lecture ou l'écriture dans le port appellera donc directement les méthodes définies dans le signal.

Exemple :

Attention Pour complètement comprendre l'exemple, vous aurez besoin de lire le chapitre sur les processus.

```
#include <systemc.h>

SC_MODULE(A)
{
    sc_in<bool> in;

    SC_CTOR(A):in("in")
    {
        SC_METHOD (trigger);
        dont_initialize();
        sensitive « in;
    }

    void trigger()
```

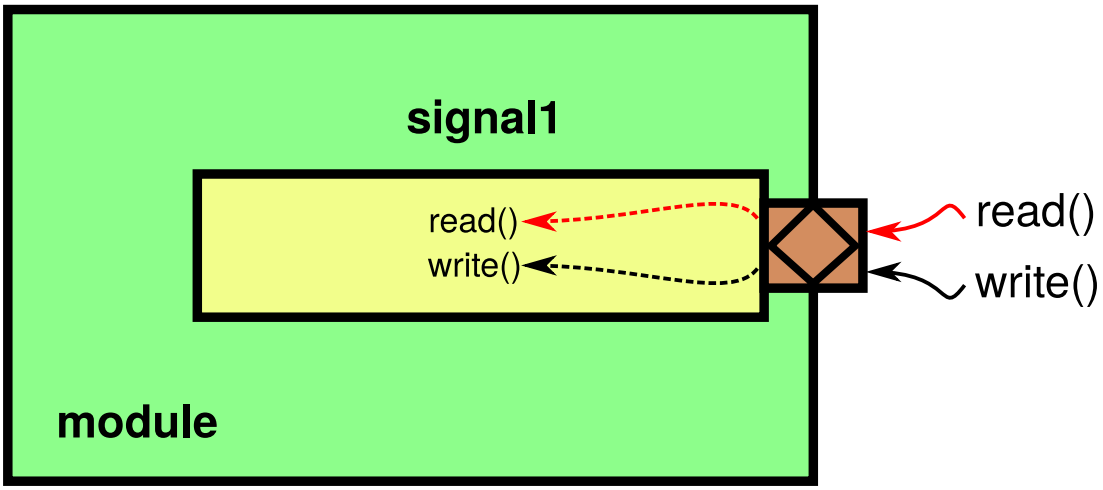

FIG. 4.1: Exports et signaux

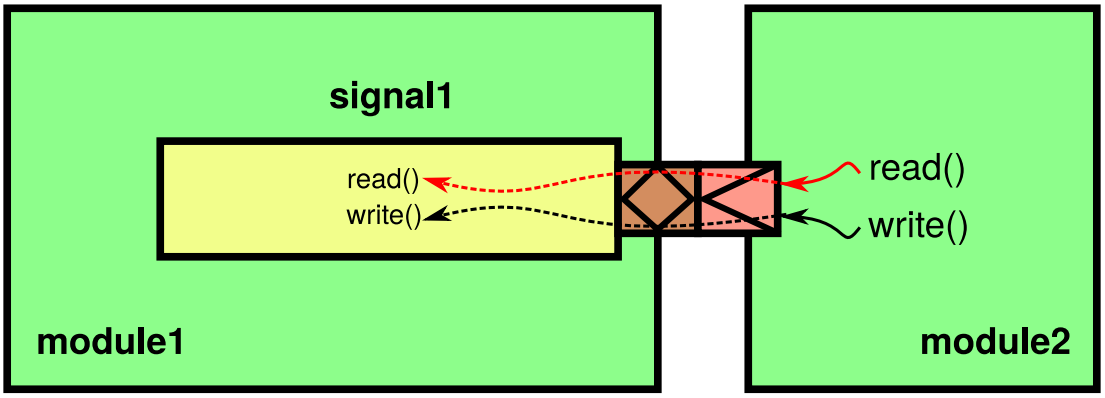

FIG. 4.2: Exports et Ports

```
    {
        cout « name() « ": triggered @" « sc_time_stamp() « endl;
    }
};


SC_MODULE (C)
{
    sc_export <sc_signal<bool> > inter;

    SC_CTOR(C): inter("inter"), sig("sig")
    {
        inter(sig);
        SC_THREAD(test);
        sig = false;
    }

    void test()
    {
       for(int i=0; i<9;i++) {
          wait(15, SC_NS);
          sig = !sig;
       }
       sc_stop();
    }

   private:
    sc_signal <bool> sig;

};

int sc_main(int argc, char* argv[])
{
    A a("a");
    C c("c");

    a.in(c.inter);

    cout
       « "let's start..." « endl
       «  "Initial: " « c.inter->read()
       « endl;

    sc_start ();
```

```
    cout « "...done" « endl
       «  "Final: " « c.inter->read()
       « endl;

    return 0;
}
```
