Introduction à SystemC

Les canaux de communication

Tarik Graba

2020-2021

Table des matières

1 Au-delà des simples signaux

Interface et canal

En SystemC on peut définir des canaux de communication complexes.

L'interface de ces canaux est séparée de la définition de leur comportement pour permettre de faire évoluer les canaux de communication sans modifier ce qui se trouve des deux cotés du canal.

La séparation entre la définition de l'interface et son comportement se fait en utilisant le concept d'interfaces de la programmation orientée objet.

- On définit dans l'interface la liste des méthodes (fonctions) que doit implémenter le canal
- On implémente dans le canal les méthodes de l'interface.

En C++, le concept d'interface utilise ce qu'on appelle des classes abstraites (virtuelles pures).

Exemple

Comment définir des interfaces en C++ ?

```
#include <iostream>
using namespace std;

class MyInterface {
public:
   // "=0" veut dire implémentation obligatoire
   virtual void hello()   = 0;
};

class SimpleImpl : virtual public MyInterface {
public:
   virtual void hello() override
   {
      cout « "Hi" « endl;
   }
};

class CplxImpl : virtual public MyInterface {
public:
```

```
    virtual void hello() override
    {
        cout <<  "Hi " << message << endl;
    }
    CplxImpl(string s): message(s) {}

private:
    const string message;
};

int main()
{
    // MyInterface   x; // ceci est une erreur car la classe est abstraite

    MyInterface * o[2];

    o[0] =  new SimpleImpl();
    o[1] =  new CplxImpl("folks");

    for (int i=0; i<2; i++)
        o[i]->hello();

    return 0;
}
```

Pour garder l'isolation entre les éléments d'une hiérarchie, les ports doivent permettre de rendre accessible les méthodes des interfaces dans un module.

Pour avoir un minimum d'interopérabilité, SystemC définit plusieurs types :

- `sc_interface`.
- `sc_prim_channel`
- `sc_port`

`sc_interface`

Définit le minimum de méthodes à implémenter pour fonctionner avec le simulateur :

- comment enregistrer un port connecté à l'interface et
- l'évènement par défaut.

Toute interface doit hériter de ce type pour pouvoir s'intégrer dans une simulation.

sc_prim_channel

Définit en plus les méthodes permettant l'interaction avec le moteur de simulation :

- affectation différée et mise à jour,
- attente (par exemple `wait()`…)

sc_port

Permet de déclarer un port pour une interface particulière. C'est une classe template dont l'un des paramètres est l'interface utilisée. Les autres paramètres correspondent au nombre de canaux qu'on peut y connecter (par défaut exactement 1).

Exemple d'utilisation

Prenons le temps de regarder le code de l'exemple.

```
#include <systemc.h>

// Une interface pour l'écriture
class tag_write_if : virtual public sc_interface
{
   public:
      virtual void write(sc_uint<8> i) =0;
};

// Une interface pour la lecture
class tag_read_if : virtual public sc_interface
{
   public:
      virtual sc_uint<16> read() =0;
};

// notre canal personnalisé, implémente les deux interfaces
// ce canal est semblable au sc_signal avec:
//    - des données de taille fixe
//    - un tag ajouté pour chaque nouvelle donnée
class tagged_bus : public tag_write_if, public tag_read_if, public sc_prim_channel
{
   sc_uint<8> tag;
   sc_uint<8> cur_d, new_d;
   sc_event   m_ev;
```

```
    public:
    SC_CTOR(tagged_bus) {
       tag   = 0;
       cur_d = 0;
       new_d = 0;
    }
    // dans tag_write_if
    virtual void write(sc_uint<8> i) {
       new_d = i;
       // on demande à ce qu'update soit appelé
       request_update();
    }
    // dans tag_read_if
    virtual sc_uint<16> read() {
       return (tag,cur_d);
    }
    // dans sc_interface
    virtual const sc_event& default_event() const {
       return m_ev;
    }
    // dans sc_prim_channel
    virtual void update() {
       if (cur_d != new_d) {
          cur_d = new_d;
          tag++;
          m_ev.notify(SC_ZERO_TIME);
       }
    }
};

SC_MODULE(W) {
    // un port en sortie n'implémente que l'interface d'écriture
    sc_port<tag_write_if> p_o;

    SC_CTOR(W) {
       SC_THREAD(loop);
    }
    void loop () {
       sc_uint<8> v = 1;
       while(v) {
          // on appelle la méthode write de l'interface
          p_o->write(v);
          v = v«1;
```

```
            wait(10, SC_NS);
        }
    }
};

SC_MODULE(R) {
    // un port en entrée n'implémente que l'interface de lecture
    sc_port<tag_read_if> p_i;

    SC_CTOR(R) {
        SC_METHOD(loop);
        // Utilise le default_event
        sensitive « p_i;
    }
    void loop () {
        // on appelle la méthode read de l'interface
        sc_uint<16> t = p_i->read();
        cout « name() «" --> tag: " « t(15,8) « " val: " « sc_bv<8>(t(7,0))« endl;
    }
};

int sc_main(int argc, char * argv[])
{

    tagged_bus b("t_bus");

    W w("writer");
    w.p_o(b);

    R r("reader");
    r.p_i(b);

    sc_start();

    return 0;
}
```

2 Et pour les sc_signal

Les sc_signal utilisent ces mêmes notions d'interfaces et canaux et de ports.

L'interface

Pour les signaux les deux interfaces suivantes sont définies :

- sc_signal_in_if<T> : interface en lecture
- sc_signal_inout_if<T> : interface en écriture

Ces deux interfaces sont définies comme des classes template, où le template T est le type des données transporté.

- sc_signal_in_if<T> : définit la méthode read() ainsi que les méthodes renvoyant les évènements.
- sc_signal_inout_if<T> : la complète en ajoutant entre autres la méthode write()

Des spécialisations de ces interfaces existent pour les types bool et sc_logic. Elles ajoutent ce qu'il faut pour les évènements sur front.

Les ports sc_in, sc_out

- Un sc_in est équivalent à un sc_port<sc_signal_in_if<T>,1>
- Un sc_out et un sc_inout sont équivalents à un sc_port<sc_signal_inout_if<T>,1>

Ces ports spécialisés ajoutent des surcharges d'opérateurs pour permettre simplement d'écrire et de lire dans le signal. Aussi, on ne peut y connecter qu'un seul signal.

Le sc_signal

Est défini comme :

```
template <class T,
          sc_writer_policy WRITER_POLICY = SC_ONE_WRITER>
class sc_signal
: public sc_signal_inout_if<T>, public sc_prim_channel
  { ... };
```

- T est le type transporté
- `WRITER_POLICY` détermine le comportement du `sc_signal` s'il est modifié par plusieurs processus. Par défaut, c'est interdit.

---

3 Des canaux standards

En plus des `sc_signal`, dans la bibliothèque, un certain nombre de canaux standards sont définis.

Les `sc_buffer<T>`

Un `sc_buffer` est équivalent à un `sc_signal` et implémente la même interface.

La seule différence, vient du fait que pour un `sc_buffer` il y a un évènement notifié à chaque écriture alors que pour un `sc_signa` il faut que la valeur change.

Travail à faire :

Écrire un exemple de code mettant en évidence la différence de comportement entre `sc_signal` et `sc_buffer`

Les `sc_fifo<T>`

Permet d'instances des fifos dont la taille est définie à l'instanciation.

Elles implémentent les interfaces :

- `sc_fifo_in_if<T>`
- `sc_fifo_out_if<T>`

Des ports spéciaux sont aussi prévus :

- `sc_fifo_in`
- `sc_fifo_out`

L'interface `sc_fifo_in_if<T>` fournit :

- `bool nb_read(&T)` : lecture non bloquante qui renvoie `true` si elle réussit.
- `void read(&T)` et `T read()` : lecture bloquante (appelle `wait()` si la fifo est vide).
- `int num_available()` : qui renvoie le nombre d'éléments disponibles dans la fifo.

- `sc_event& data_written_event()` : renvoie une référence vers évènement notifié en cas d'écriture.

L'interface `sc_fifo_out_if<T>` fournit :

- `bool nb_write(&T)` : écriture non bloquante qui renvoie `true` si elle réussit.
- `void write(&T)` : écriture bloquante (appelle `wait()` si la fifo est pleine)
- `int num_free()` : renvoie le nombre d'éléments pouvant être écrits dans la fifo.
- `sc_event & data_read_event()` : une référence vers un évènement notifié en cas de lecture.

Attention comme les méthodes bloquantes de lecture et d'écriture font appel à `wait()` pour suspendre le processus qui les appelle, il faut donc prendre des précautions quand on les utilise dans des `SC_METHOD`.

Le constructeur d'une `sc_fifo<T>` prend en argument la taille de la fifo. La taille par défaut est de **16**. De plus l'opérateur d'affectation a été surchargé pour appeler les méthodes **bloquantes** de lecture et d'écriture.

Exemple :

```
// une fifo de 16 éléments
sc_fifo<int> A;
// une fifo de 32 éléments
sc_fifo<int> B(32);

// deux écritures bloquantes
A.write(1);
B = 1;

// deux lectures bloquantes
int x;
x = A;
x = B.read();
```

Exemple plus complet :

```
#include <systemc.h>
SC_MODULE(A) {
   sc_fifo_out<int> out;
   void loop() {
      int i = 0;
      for(;;){
         // write est obligatoire
         out.write(i);
         i++;
         wait(33, SC_NS);
      }
   }
   SC_CTOR(A) { SC_THREAD(loop); }
};
```

```
SC_MODULE(B) {
   sc_fifo_in<int> in;
   void loop() {
      wait(300, SC_NS);
      for(;;){
         // in.read() appelle in->read()
         cout « "Lecture 1 : " « in->read() « "@ " « sc_time_stamp() « endl;
         cout « "Lecture 2 : " « in.read()  « "@ " « sc_time_stamp() « endl;
         wait(55, SC_NS);
      }
   }
   SC_CTOR(B) { SC_THREAD(loop); }
};

int sc_main(int, char **) {
   // une fifo de 10 entiers
   sc_fifo<int> fifo(10);

   A a("modA");
   a.out(fifo);

   B b("modB");
   b.in(fifo);

   sc_start(300,SC_NS);
   cout
      « "contenu de la fifo @" « sc_time_stamp() « endl
      « fifo « endl;

   sc_start(1,SC_US);
   cout
      « "contenu de la fifo @" « sc_time_stamp() « endl
      « fifo « endl;

   return 0;
}
```

Travail à faire :

Écrire le code de deux modules s'échangeant des Pixels à travers une fifo. Les modules devront fonctionner à des cadences différentes tout en étant synchrones à une horloge `clk` générée au niveau supérieur. Par exemple :

- Producteur : envoie, une à une, 100 données en attendant un cycle entre chaque envoie puis s'arrête pendant un nombre aléatoire de cycles compris entre 0 et 32.

- Consommateur : lit une donnée puis attend 2 cycles si la somme des composantes du pixel lu est paire ou 3 cycles elle est impaire.
- Observez le comportement pour différentes profondeurs de fifo.

Faites deux implémentations du producteur, l'une utilisant des SC_THREAD ou SC_CTHREAD et l'autre utilisant des SC_METHOD.

Les sc_mutex et sc_semaphore

La bibliothèque définit d'autres canaux standards permettant la synchronisation de SC_THREAD pour l'accès à des ressources partagées.

Le sc_mutex fournit les méthodes :

- `int lock()` : pour verrouiller le mutex. S'il est déjà verrouillé, on est mis en attente (`wait()` est appelé). Cette méthode renvoie toujours **0**.
- `int trylock()` : pour essayer verrouiller le mutex. Cette méthode renvoie **0** en cas de succès, **-1** sinon.
- `int unlock()` : pour libérer le mutex. Cette méthode renvoie **0** en cas de succès, **-1** sinon (i.e. le mutex n'est pas verrouillé ou il appartient à un autre thread).

Le sc_semaphore prend en argument de constructeur le nombre de *sémaphores* qu'on peut prendre. Si ce nombre est égal à un, son comportement est équivalent à un sc_mutex.

Le sc_semaphore fournit les méthodes :

- `int wait()` : si le nombre de sémaphores restants est supérieur à 0, le décrémenter et retourner, sinon, le processus est suspendu (`wait()`) Cette méthode renvoie toujours **0**.
- `int trywait()` : si le nombre de sémaphores restants est supérieur à 0, le décrémenter et retourner **0**, sinon, retourne **-1**.
- `int post()` : incrémente le nombre de sémaphores. Cette méthode renvoie toujours **0**.
- `int get_value()` : renvoie le nombre de sémaphores disponibles.

Travail à faire :

Écrire le code d'un module dans lequel **deux** SC_THREAD concurrents s'exécutent à tour de rôle en utilisant un sc_mutex.

Un sc_mutex suffirait-il si on avait **trois** processus ?

---

4 sc_channel

SystemC définit la notion de canal hiérarchique. Un canal hiérarchique n'est en réalité qu'un `sc_module` qui, en utilisant entre autres des `sc_export`, peut se présenter comme un canal de communication complexe.

On peut donc y trouver des processus et modéliser les comportements du canal de communication.

Pour définir explicitement un canal hiérarchique on utilisera la classe `sc_channel` qui n'est qu'un alias de la classe `sc_module`.

Travail à faire :

En utilisant la classe `sc_channel` écrire le code d'un générateur d'horloge. Ce canal contiendra un `sc_signal` qui pourra être connecté à l'entrée d'un module de test à travers un `sc_export`.
