Introduction à SystemC

Évènements et processus

Tarik Graba

P4 2018-2019

Télécom ParisTech

Table des matières

1 Les évènements

Les `sc_event`

En SystemC, les évènements sont des objets de type `sc_event`.

Quand un évènement est instancié, il s'ajoute à la liste des évènements "surveillés" par le simulateur.

Les évènements peuvent ensuite être "notifiés".

Cette notification déclenchera des actions au niveau du simulateur.

On a rarement besoin de les instancier directement dans le code. Ils sont généralement instanciés par les objets qui en ont besoin.

La notification des `sc_event`

Les `sc_event` implémentent les méthodes :

```
void notify();
void notify( const sc_time& );
void notify( double, sc_time_unit );
```

Sans argument, elle permet de notifier un évènement immédiatement.

Sinon, la notification est programmée (schedule) pour avoir lieu plus tard. L'argument correspondant alors au délai par rapport au temps actuel de la simulation.

Si le temps fourni en argument est nul (ou égal à SC_ZERO_TIME) la notification se fera à la fin du delta.

Les évènements pour un `sc_signal`

La classe `sc_signal` contient un pointeur vers un `sc_event`.

À l'élaboration, si besoin, il est alloué et ajouté à la liste des évènements du simulateur.

Quand le signal **change de valeur**, l'évènement est notifié pour la fin du delta.

Une version simplifiée serait :

```
template<typename T>
class simple_sc_signal
{
   T cur_val;
   T new_val;
   // L'évènement est initialisé à NULL dans le constructeur
   // Si besoin durant la phase d'élaboration il sera alloué
   sc_event * m_event;
   public:

   const T& read() const {
      return cur_val;
   }
   void write( const T& v){
      new_val = v;
   }
   void update( ){
      if (cur_val != new_val) {
         cur_val = new_val;
         if (m_event)
            m_event->notify(SC_ZERO_TIME);
      }
   }
};
```

En réalité c'est plus compliqué que ça. Les évènements ne sont vraiment créés durant la phase d'élaboration (dynamiquement) que si on n'en a besoin. Et la notification ne se fait que dans ce cas.

Pour un signal on peut récupérer une référence vers l'évènement en utilisant l'une des deux méthodes :

- `value_changed_event()`
- `default_event()`

Qui pour un `sc_signal` font référence au même évènement.

***Cas particulier les* `sc_signal<bool>`**

Pour un `sc_signal<bool>` en plus des évènements génériques, on peut récupérer des évènements particuliers en cas de fronts montants ou descendants. Les méthodes permettant d'accéder à ces évènements sont :

- posedge_event()
- negedge_event()

---

2 Les processus

Les processus

En SystemC il existe deux types principaux de processus :

- Les SC_THREAD
- Les SC_METHOD

Déclaration d'un processus

Les processus sont des méthodes du `sc_module`.

Ces méthodes particulières doivent *enregistrées* en utilisant l'une des macros suivantes dans le constructeur du module :

- SC_THREAD
- SC_CTHREAD (cas particulier de SC_THREAD)
- SC_METHOD

Pour pouvoir enregistrer une méthode d'un `sc_module` comme processus, elle :

- ne doit pas prendre d'arguments,
- doit être de type `void`.

Les SC_THREAD

- Les SC_THREAD sont des threads indépendants qui sont lancés au début de la simulation (au premier `sc_start()`) dans le contexte du simulateur.
- C'est du multitâche collaboratif, les SC_THREAD doivent se mettre en veille pour permettre à la simulation d'avancer.
- En veille, ils peuvent être réveillés par la notification d'un évènement.

Attention

Les SC_THREAD ne sont pas des threads du système. Ce sont des threads exécutés dans le contexte du simulateur qui lui reste vu comme un processus unique.

Quand un SC_THREAD se termine (on arrive à la fin de la fonction, ou à un `return`) il n'est plus relancé. Si on veut maintenir en *vie* un SC_THREAD on *doit* l'implémenter sous la forme d'une *boucle infinie*.

Déclarer un SC_THREAD

La macro SC_THREAD permet de faire cela.

Elle doit être utilisée dans le constructeur du `sc_module`. Elle prend comme unique argument la méthode associée.

L'exemple suivant montre comment déclarer un SC_THREAD :

```
SC_MODULE(foo)
{
   // La déclaration d'une méthode interne du module
   void bar();

   // Le constructeur du module
   SC_CTOR(foo)
   {
      // Enregistrement de la méthode 'bar' comme sc_thread
      SC_THREAD(bar);
   }
};
```

La mise en veille d'un SC_THREAD

La mise en veille d'un SC_THREAD se fait en appelant la méthode `wait()`.

En fonction des arguments passés, le SC_THREAD est alors mis en veille :

- pour un temps donné,
- jusqu'à ce qu'un évènement soit notifié.

- Si l'argument passé à `wait` est un temps (sous la forme d'un `sc_time` ou d'un couple `double` plus unité) alors le processus est mis en veille immédiatement et son *reveil* est programmé dans l'échéancier.
- Si l'argument est un évènement, le SC_THREAD est mis en veille jusqu'à la notification de ce processus.
- Si `wait` est appelé sans argument, le réveil se fera sur un des évènements de la liste de sensibilité du SC_THREAD (voir section suivante).

Exemples

```
sc_time T(133,SC_NS);

SC_MODULE(foo)
{
   sc_in<bool> a;
   sc_in<int>  b;

   void bar()
   {
      // On attend un temps
      wait(254,SC_NS);
      ...
      // Ici aussi, on attend un temps
      wait(T);
      ...
      // On attend un front montant de a
      wait(a.posedge_event());
      ...
      // On attend que b change (si le port est connecté à un signal)
      wait(b.default_event());
      ...
   }

   SC_CTOR(foo)
   {
      SC_THREAD(bar);
   }
};
```

La liste de sensibilité d'un SC_THREAD

Pour un SC_THREAD on peut définir une liste de sensibilité statique en utilisant la construction `sensitive`

```
sensitive << e1 << e2 ;
```

`sensitive` est un champ des `sc_module` qui permet de définir la liste de sensibilité d'un processus. L'opérateur de flux « a été surchargé pour ajouter un élément à la liste de sensibilité.

La liste de sensibilité ne concerne que le dernier processus déclaré.

Exemple

```
SC_MODULE(foo)
{
   sc_in<bool> a;
   sc_in<int>  b;
   sc_in<int>  c;

   SC_CTOR(foo)
   {
      // Le thread bar est sensible aux évènements sur a, b et c
      SC_THREAD(bar);
      sensitive << a << b ;
      sensitive << c ;
      // Le thread lab est sensible à l'évènement "front montant" de a
      SC_THREAD(lab);
      sensitive << a.pos();
   }

   void bar()
   {
      ...
      // attendre un évènement sur a, b ou c
      wait();
      ...
   }

   void lab()
   {
      ...
      // attendre un front montant de "a"
      wait();
      ...
   }
};
```

Une fois la liste de sensibilité définie, on peut utiliser la méthode `wait()` sans argument. Le `SC_THREAD` est mis en veille en attente d'un des évènements de sa liste de sensibilité.

La liste de sensibilité peut contenir des ports en entrée ou des signaux internes du module. C'est l'évènement par défaut (`default_event()`) du signal (ou du signal connecté au port) qui sera enregistré dans la liste de sensibilité du processus.

Cas particulier

Pour les ports de types `sc_in<bool>`, on peut vouloir restreindre les évènements qui déclenchent le processus aux fronts montants ou ascendants. Dans ce cas, on peut explicitement y faire référence en utilisant les méthodes suivantes :

```
...
sc_in<bool> c;

...
SC_THREAD(...)
sensitive << c.pos(); // sensible aux fronts montants sur c
sensitive << c.neg(); // sensible aux fronts descendants sur c
```

La remise à zéro d'un `SC_THREAD`

Pour pouvoir simuler la remise à zéro (`reset`) d'un bloc matériel ou logiciel, on doit pouvoir durant la simulation demander la réexécution à partir du début d'un processus.

Il existe deux méthodes :

- `reset_signal_is`
- `async_reset_signal_is`

qui permettent de définir des signaux de remise à zéro synchrone ou asynchrone.

Le signal utilisé pour la remise à zéro *doit* être de type `bool`. Il peut être déclaré en interne ou être connecté à un port du module.

Elles prennent deux arguments, le signal et la polarité (positive ou négative) pour laquelle le reset est actif sous la forme d'un booléen (`true`,`false`).

```
// r est un signal de remise à zéro synchrone actif sur niveau haut
reset_signal_is(r,true);
// nrst est un signal de remise à zéro asynchrone actif sur niveau bas
async_reset_signal_is(nrst,false);
```

Comme `sensitive` ces méthodes agissent sur le dernier processus déclaré.

Différence entre synchrone et asynchrone

Un signal de remise à zéro asynchrone fait automatiquement partie de la liste de sensibilité. S'il change d'état son effet est immédiat.

Un signal de remise à zéro synchrone ne fait pas partie de la liste de sensibilité. Son effet ne sera visible que quand le processus sera réactivé.

Comment s'en servir dans un SC_THREAD

L'utilisation des signaux de remise à zéro sous-entend un certain style de codage comme le montre l'exemple suivant :

```
SC_MODULE(foo)
{
   sc_in<bool> c;
   sc_in<bool> r;

   void bar()
   {
      // Ce qu'on fait au début et à chaque fois que r passe à true
      ...
      // Fonctionnement normal dans une boucle infinie
      for(;;)
      {
         // ce qu'on fait à chaque front montant de "c"
         ...
      }
   }

   SC_CTOR(foo)
   {
      SC_THREAD(bar);
      sensitive << c.pos();
      async_reset_signal_is(r,true);
   }
};
```

La première partie du processus est exécutée au démarrage initial du processus puis à chaque fois que le signal de remise à zéro est actif. En suite, le processus entre dans une boucle infinie dans laquelle on retrouve le fonctionnement "normal".

Attention, si le processus se termine (fin de la fonction, return…) il ne pourra plus être redémarré même si un signal de remise à zéro a été défini.

Spécialisation : les SC_CTHREAD

Cas particulier de SC_THREAD pour modéliser le logique synchrone.

On *doit* y associer un signal d'horloge qui ferra automatiquement partie de sa liste de sensibilité.

Exemple

```
SC_MODULE(foo)
{
   sc_in<bool> clk;

   // La déclaration d'une méthode interne du module
   void bar();

   // Le constructeur du module
   SC_CTOR(foo)
   {
      // Enregistrement de la méthode 'bar' comme sc_cthread
      // Elle sera déclanchée sur les fronts montants de clk
      SC_CTHREAD(bar, clk.pos());
   }
};
```

Spécialisation : wait dans un SC_CTHREAD

Dans un SC_CTHREAD, la méthode `wait()` peut prendre un argument entier pour indiquer le nombre de périodes d'horloge à attendre.

Exemple

```
SC_MODULE(foo)
{
   sc_in<bool> clk;

   int lat;

   void bar()
   {
      // On un cycle d'horloge
```

```
        wait();
        ...
        // Ici aussi
        wait();
        ...
        // On attend 33 cycles d'horloge
        wait(33);
        ...
        // On attend "lat" cycles
        wait(lat);
        ...
    }

    SC_CTOR(foo)
    {
        SC_CTHREAD(bar, clk.pos());
    }
};
```

Pour le reste le comportement d'un SC_CTHREAD est équivalent à celui du SC_THREAD.

Les SC_METHOD

Second type de processus, les SC_METHOD sont équivalentes aux processus des autres HDL. Elles permettent de faire d'écrire des représentations "RTL".

Contrairement aux SC_THREAD les SC_METHOD ne sont pas autonomes. Elles sont exécutées par le scheduler quand un évènement de leur liste de sensibilité est notifié.

Elles doivent s'exécuter entièrement et rendre la main au scheduler sans quoi la simulation est bloquée. Appeler la méthode `wait()` dans une SC_METHOD est donc interdit.

Une SC_METHOD est équivalente aux processus `always` de Verilog/SystemVerilog.

Pour l'enregistrer on utilise la macro SC_METHOD.

Exemple

```
SC_MODULE(foo)
{
    // La déclaration d'une méthode interne du module
    void bar();
```

```
   // Le constructeur du module
   SC_CTOR(foo)
   {
      // Enregistrement de la méthode 'bar' comme sc_method
      SC_METHOD(bar);
   }
};
```

La liste de sensibilité d'une SC_METHOD

La déclaration de la liste de sensibilité des SC_METHOD utilise aussi la construction `sensitive`

Exemple

```
SC_MODULE(foo)
{
   sc_in<int>  a;
   sc_in<int>  b;
   sc_out<int> c;

   SC_CTOR(foo)
   {
      // La méthode bar est sensible aux évènement sur a et b
      SC_METHOD(bar);
      sensitive << a << b ;
   }

   // La méthode calcule la somme de a et de b et modifie c
   // puis rend la main au scheduler
   void bar()
   {
      c = a + b;
   }
};
```

Gestion de la remise à zéro dans une SC_METHOD

La remise à zéro des SC_METHOD peut aussi utiliser les méthodes `reset_signal_is` et `async_reset_signal_is`.

La méthode `async_reset_signal_is` ajoute aussi le signal de reset à la liste de sensibilité.

Les SC_METHOD sont toujours exécutées du début à la fin. Le teste de la condition de reset se fait alors dans le code de la méthode.

Exemple

```
SC_MODULE(foo)
{
   sc_in<bool> c;
   sc_in<bool> r;

   void bar()
   {
      // Ce qu'on fait au début et à chaque fois que r passe à true
      if (r)
      {
         ...
      }
      // ce qu'on fait à chaque front montant de "c"
      else
      {
         ...
      }
   }

   SC_CTOR(foo)
   {
      SC_METHOD(bar);
      sensitive << c.pos();
      reset_signal_is(r,true);
   }
};
```

La méthode `dont_initialize`

Par défaut, tous les processus sont exécutés au début de la simulation avant le moindre évènement. Si ce comportement n'est pas désiré, il faut appeler la méthode `dont_initialize()` après l'enregistrement du processus.

Le premier lancement du processus se fait alors au premier évènement notifié.

La méthode `dont_initialize()` agit sur la dernière méthode enregistrée.

Pour avoir un comportement équivalent à celui d'une simulation Verilog/SystemVerilog, il faut appeler `dont_initialize()` pour toutes les `SC_METHOD`.

```
SC_MODULE(foo)
{
   sc_in<int>  a;
   sc_in<int>  b;
   sc_out<int> c;

   SC_CTOR(foo)
   {
      // La méthode bar est sensible aux évènement sur a et b
      SC_METHOD(bar);
      sensitive << a << b ;
      // attendre le premier évènement avant de faire le premier calcul
      dont_initialize();
   }

   // La méthode calcule la somme de a et de b et modifie c
   // puis rend la main au scheduler
   void bar()
   {
      c = a + b;
   }
};
```

Pour les `SC_THREAD`, qui sont supposés être autonomes et qui n'ont pas de liste de sensibilité, appeler `dont_initialize()` fera que le thread ne démarrera jamais.

```
SC_MODULE(foo)
{
   sc_out<int> c;

   SC_CTOR(foo)
   {
```

```
      SC_THREAD(bar);
      dont_initialize();
   }

   //  /! A cause du dont_initialize, ce thread ne démarrera jamais !!!
   void bar()
   {
      c = 0;
      for(;;)
      {
         wait(100, SC_NS);
         c = c + 1;
      }
   }
};
```

Résumons

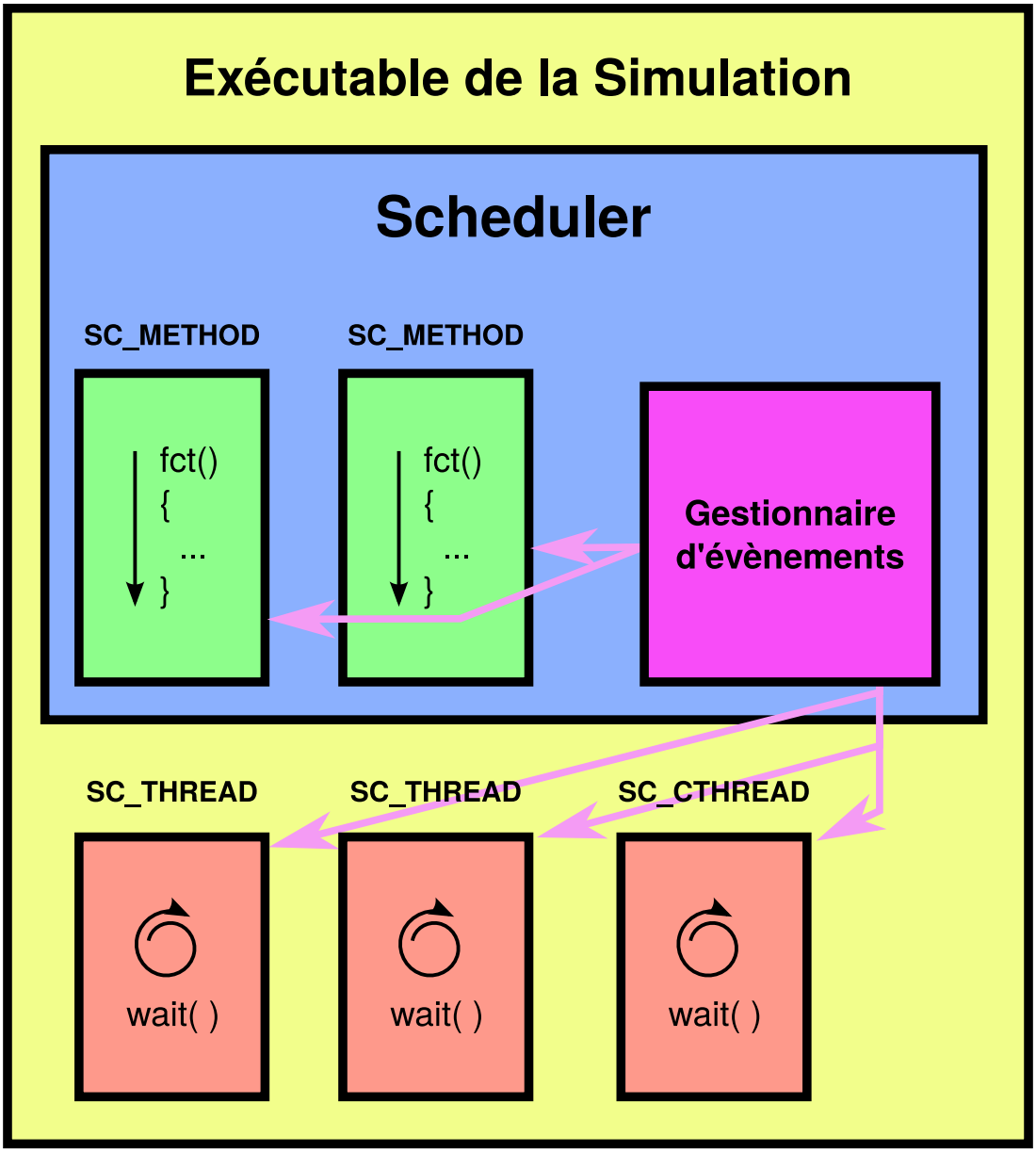

FIG. 2.1: Les processus

3 Travail à faire

Modéliser

En reprenant le classe `Pixel` définie dans la section sur les signaux écrivez les modèles suivants :

- Deux modules combinatoires faisant la somme saturante de deux pixels.
  - Le premier en utilisant une `SC_METHOD`
  - Le second en utilisant un `SC_THREAD`
- 3 modules séquentiels qui génère successivement, à la cadence d'une horloge externe et de façon cyclique, toutes les couleurs possibles :
  - Vous utiliserez `SC_METHOD`, `SC_THREAD` et `SC_CTHREAD`
  - L'état initial sera la couleur `(0,0,0)` et pourra être forcé grâce à un signal de remise à zéro asynchrone.

Intégrez ces modules dans un `sc_main` permettant de vérifier le fonctionnement et générer les chronogrammes.