Introduction à SystemC

Simulation et signaux

Tarik Graba

2020-2021

Table des matières

1 Le simulateur

Le simulateur événementiel

Le simulateur événementiel fait partie de la bibliothèque SystemC.

L'implémentation de référence fournie contient donc tout le nécessaire pour faire des simulations.

Faire avancer la simulation

En SystemC on contrôle l'avancée de la simulation dans le code lui-même.

La fonction principale pour lancer le simulateur est `sc_start`.

- Elle ne peut être appelée qu'à partir de `sc_main` (ou dans une fonction appelée dans `sc_main`)
- Elle peut prendre en argument un temps
  - dans ce cas, la simulation avance du temps indiqué
- Sans argument, la simulation est lancée indéfiniment
  - tant qu'il y a de l'activité,
  - tant que `sc_stop` n'a pas été appelé.

L'argument fourni à `sc_start` peut être soit :

- un `sc_time`,
- un couple `double` plus unité de temps

Pour faire avancer la simulation d'un delta (un cycle de simulation) sans faire avancer le temps, il suffit de passer à `sc_start` la constante `SC_ZERO_TIME`.

```
#include <systemc.h>

int sc_main (int argc, char * argv[])
{
   sc_time T(0.333, SC_NS);

   cout « "On commence à " « sc_time_stamp() « " " « sc_delta_count() « endl;

   sc_start(T);
   cout « " ... " « sc_time_stamp() « " " « sc_delta_count() « endl;
```

```
    sc_start(0.333,SC_NS);
    cout « " ... " « sc_time_stamp() « " " « sc_delta_count() « endl;

    sc_start(SC_ZERO_TIME);
    cout « " ... " « sc_time_stamp() « " " « sc_delta_count() « endl;

    sc_start();
    cout « " ... " « sc_time_stamp() « " " « sc_delta_count() « endl;

    return 0;
}
```

Notes :

- La fonction `sc_time_stamp` renvoie la valeur actuelle du temps dans la simulation
- La fonction `sc_delta_count` renvoie la valeur du compteur de cycles de simulation (delta)

Dans cet exemple, le nombre de cycles de simulation n'avance pas car il n'y a aucun *processus*. Les processus seront présentés plus tard dans ce cours.

Visualisation

Il est souvent intéressant de visualiser l'état de la simulation au cours du temps. On peut toujours utiliser les fonctions d'impression standard de C++ car les objets de la bibliothèque SystemC les supportent.

Pour une visualisation graphique, SystemC fournit les fonctions nécessaires à la génération de chronogrammes au format VCD.

Le format Value Change Dump (VCD) est un format standard pour enregistrer au format ASCII les changements d'états dans une simulation. Il fait partie du standard Verilog/SystemVerilog.

Dans un fichier VCD ne sont sauvegardés que les changements d'états des signaux et des variables. On parle généralement de fichier de *trace*.

Traces VCD en SystemC

Les éléments principaux définis dans la bibliothèque pour générer des traces sont :

- Le type `sc_trace_file` pour gérer le fichier de trace
- La fonction `sc_create_vcd_trace_file` pour créer le fichier

- La fonction `sc_trace` pour ajouter un objet aux objets à tracer
- La fonction `sc_close_vcd_trace_file` pour fermer le fichier

La génération des traces se fait ensuite automatiquement durant une simulation. On peut ensuite les voir avec un outil dédié.

Par défaut, la fonction `sc_trace` permet de tracer les types standards de C++ ainsi que les types SystemC. Nous verrons par la suite qu'elle permet de tracer d'autres objets définis dans SystemC et qu'on peut la surcharger pour suivre des types personnalisés.

L'exemple suivant montre comment utiliser ce type de traces :

```
#include <systemc.h>

int sc_main (int argc, char * argv[])
{
   // Un pointeur sur l'objet qui permet de gérer les traces
   sc_trace_file *trace_f;

   // Cette fonction crée l'objet
   // L'argument est le nom du fichier qui sera créé.
   // L'extension .vcd est ajoutée automatiquement
   trace_f = sc_create_vcd_trace_file ("my_simu_trace");
   // On peut aussi préciser l'unité de temps dans le fichier vcd
   trace_f->set_time_unit(1,SC_NS);

   bool t = false;

   // Ajoute la variable t aux éléments à tracer
   // Les arguments de la fonction sont:
   //    - le pointeur vers le fichier de trace
   //    - la variable/objet à tracer
   //    - le nom dans dans le fichier généré
   sc_trace(trace_f, t, "t");

   // La simulation
   sc_start(10,SC_NS);
   t = !t;
   sc_start(10,SC_NS);
   t = !t;
   sc_start(10,SC_NS);
   t = !t;
   sc_start(10,SC_NS);
```

```
    // Ferme le fichier de trace
    // ne peut êter fait qu'à la fin de la simulation
    sc_close_vcd_trace_file(trace_f);

    return 0;
}
```

À la fin de l'exécution de la simulation, un fichier nommé `my_simu_trace.vcd` est créé.

Il existe plusieurs outils permettant de visualiser des traces au format VCD parmi lesquels l'outil opensource `gtkwave`. Il fait généralement partie des *paquets* disponibles dans la majorité des distributions GNU-Linux.

Travail à faire

Modifier le code de l'exemple pour générer les traces d'une variable entière positive qui s'incrémente de `0` à `N` toutes les `10ns`. Le maximum, `N`, doit pouvoir être modifié sans recompiler l'exécutable et ne dépassera jamais 255.

---

2 Les signaux

Au-delà de simples signaux

Dans une simulation événementielle, pour garantir le déterminisme, nous avons besoin de *signaux*. Les signaux permettent les affectations différées.

Un signal, possède deux valeurs :

- la valeur actuelle,
- la valeur à la fin du cycle de simulation.

Le type `sc_signal<T>`

SystemC définit le type templaté `sc_signal`. Le paramètre de template est un type, permettant ainsi de créer des signaux *transportant* tout type de donnée.

```
sc_signal<bool> x;          // un signal transportant un booléen
sc_signal<int>  y;          // un signal transportant un entier
sc_signal<sc_int<14> >  z; // un signal transportant un entier
                            // SystemC sur 14 bits
```

L'implémentation `sc_signal<T>`

- 2 valeurs : courante et future
- 3 méthodes `read`, `write` et `update`

Comment un `sc_signal` *est-il implémenté ?*

C'est une classe C++ templatée par un type T et contenant deux variables de ce type :

- `cur_val` : la valeur actuelle du signal,
- `new_val` : la valeur à la fin du cycle de simulation.

Cette classe implémente deux méthodes `read` et `write` telles que :

- `write` : ne modifie que la valeur future `new_val`
- `read` : renvoie la valeur actuelle `cur_val`

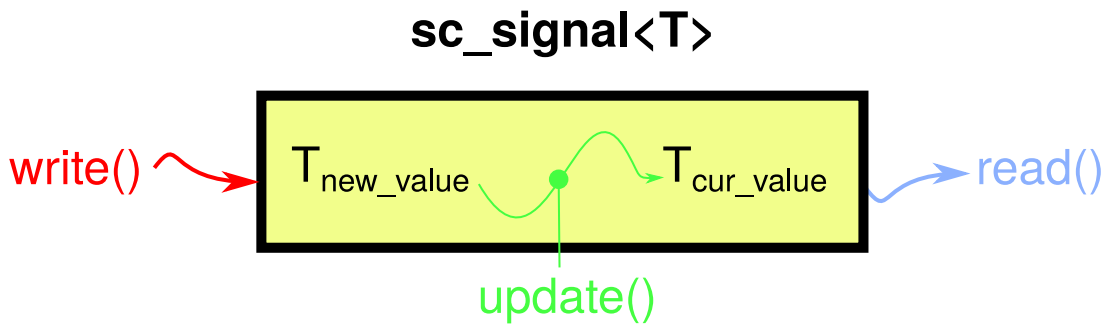

FIG. 2.1: Un sc_signal

- update : met à jour la valeur actuelle

Une version simplifiée serait :

```
template<typename T>
class simple_sc_signal
{
   T cur_val;
   T new_val;
   public:

   const T& read() const {
      return cur_val;
   }
   void write( const T& v){
      new_val = v;
   }
   void update( ){
      cur_val = new_val;
   }
};
```

ATTENTION C'est une version très simplifiée.

De plus, l'opérateur d'affectation = est surchargé pour que toute affectation vers un sc_signal appelle la méthode write. Et toute référence à un sc_signal appelle la méthode read.

On peut générer des traces de signaux, tant que le type transporté le supporte.

L'exemple suivant montre comment évolue un signal dans une simulation SystemC. Notez aussi la différence par rapport à une variable standard.

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

```
int sc_main (int argc, char * argv[])
{
   sc_time T(0.333, SC_NS);

   sc_signal<int> i;
   int j = 0;

   cout « ”init: ” « sc_time_stamp() « ” j = ” « j «” et i = ” « i « endl;

   i = 33;
   j = 33;
   cout « ”--> @ ” « sc_time_stamp() « ” j = ” « j «” et i = ” « i « endl;

   sc_start(T);
   cout « ”--> @ ” « sc_time_stamp() « ” j = ” « j «” et i = ” « i « endl;

   i = 44;
   j = 44;
   cout « ”--> @ ” « sc_time_stamp() « ” j = ” « j «” et i = ” « i « endl;

   sc_start(SC_ZERO_TIME);
   cout « ”--> @ ” « sc_time_stamp() « ” j = ” « j «” et i = ” « i « endl;

   return 0;
}
```

De plus, un `sc_signal` peut être nommé en lui passant une chaîne de caractère comme argument de constructeur. Si cet argument n'est pas donné, un nom *générique* est créé automatiquement. Des mécanismes existent pour garantir l'unicité de ce nom. Ce nom peut être récupéré en appelant la méthode name

```
#include <systemc.h>

int sc_main (int argc, char * argv[])
{
   sc_signal<int> i(”i”);
   sc_signal<int> j(”je m'appelle j”);
   sc_signal<int> x;

   cout « ”--> ” « i.name() « ”: ” « i « endl;
   cout « ”--> ” « j.name() « ”: ” « j « endl;
   cout « ”--> ” « x.name() « ”: ” « x « endl;

   return 0;
}
```

Remarque Par défaut, un signal ne supporte qu'un seul *écrivain*. C'est le cas ici, puisque qu'on ne modifie le signal que dans `sc_main`.

Quels types peut-on transporter ?

Le type T peut être :

- Un type standard de C++ (`bool`, `int`, ...)
- Un des types définis par SystemC (`sc_logic`, `sc_int`, ...)
- Tout type respectant certaines conditions

Pour qu'une classe puise être utilisée comme template d'un `sc_signal` il faut qu'elle remplisse les conditions suivantes :

- pouvoir faire des tests d'égalité en utilisant l'opérateur ==,
- pouvoir faire des affectations en utilisant l'opérateur =,
- avoir un constructeur par défaut (sans fournir d'arguments),
- supporter l'opérateur de flux « pour pouvoir afficher sa valeur,
- supporter la fonction de trace (`sc_trace`)

Exemple :

```
#include <systemc.h>

// un type utilisateur
struct pt_t {
   int i;
   int j;
   // un constructeur particulier avec des valeurs par défaut
   pt_t( int _i=0, int _j=1): i(_i), j(_j) { }

   bool operator == (const pt_t &other) const {
      return (i == other.i) && (j == other.j);
   }
   // On doit pouvoir imprimer la valeur d'un objet de ce type
   // l'opérateur « est un opérateur de la classe std::ostream
   friend ostream& operator « ( ostream& o, const pt_t& P ) {
      o « "{" « P.i « "," « P.j « "}" ;
      return o;
   }
};
```

```
// surcharge de la fonction sc_trace pour le type utilisateur
void sc_trace( sc_trace_file* _f, const pt_t& _foo, const std::string& _s ) {
   sc_trace( _f, _foo.i, _s + "_i" );
   sc_trace( _f, _foo.j, _s + "_j" );
}

// Le test
int sc_main (int argc, char * argv[])
{
   sc_signal<pt_t> P;
   cout « "--> @ " « sc_time_stamp() « " P = " « P « endl;

   // affectation au signal
   P = pt_t(33,22);
   cout « "--> @ " « sc_time_stamp() « " P = " « P « endl;

   sc_start(1,SC_NS);
   cout « "--> @ " « sc_time_stamp() « " P = " « P « endl;

   return 0;
}
```

Travail à faire

- Écrivez une structure `Pixel` qui encapsule trois composantes nonn signées (R,G,B) au format (5 :6 :5).
- Faites évoluer un signal transportant un `Pixel` en incrémentant successivement chaque composante.
- Générez-les *traces* montrant cette évolution

Un signal d'horloge avec sc_clock

Le type `sc_clock` est une spécialisation de `sc_signal<bool>`. Quand la simulation avance, la valeur d'une `sc_clock` change cycliquement en fonction des paramètres qui lui ont été passés.

```
   // Une horloge de periode 10ns
   sc_clock clk("clock",10,SC_NS);
```

On ne peut pas écrire dans un objet de type `sc_clock`.

À la construction, on peut fournir les paramètres suivants :

- un nom,

- une période,
- un rapport cyclique,
- l'instant du premier front,
- le sens du premier front.

Les temps peuvent être donnés sous forme de `sc_time` ou d'un `double` plus une unité.

```
#include <systemc.h>

int sc_main(int argc, char * argv[]) {
   // une horloge de période 10ns
   sc_clock ck1("ck1",10,SC_NS);

   sc_time T(25,SC_NS);
   // une horloge de période T
   sc_clock ck2("ck2",T);

   // une horloge de période 10ns avec un rapport cyclique de 25%
   // qui commence à l'instant 150ns par un front descendant
   sc_clock ck3("ck3",10,SC_NS,0.25,150,SC_NS,false);

   double D = 0.4;
   sc_time ST(200,SC_NS);
   bool first_edge = true;
   // une horloge de période T avec un rapport cyclique D
   // qui commence à l'instant ST par un front montant
   sc_clock ck4("ck4",T,D,ST,first_edge);

   return 0;
}
```

Travail à faire

Afficher les traces de ces différents signaux d'horloge.

Résolution de conflits

Pour modéliser un bus 3 états on doit utiliser des signaux particuliers :

- `sc_signal_resolved` équivalent à un `sc_signal<sc_logic>`
- `sc_signal_rv<N>` équivalent à un `sc_signal<sc_lv<N> >`

Ces signaux supportent plusieurs *écrivains* et contient des fonctions de résolution.

La fonction de résolution permet de calculer la valeur du signal si 2 *écrivains* modifient en même temps la valeur du signal.

La fonction utilisée est la suivante :

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