Ecole Nationale Supérieure d'Ingénieurs de Tunis

Les Systèmes Embarqués

Chapitre4. Modélisation et simulation d'un système embarqué en SystemC

Plan

1. Les types de données
2. Les interfaces
3. Les canaux de communication
4. Les modules
5. Les ports
6. Les processus
7. Le contrôle des simulations

1. Les types de données

- Les modèles SystemC peuvent manipuler différents types de données :
  - Les types natifs C++ : char, int, short, long, long long, …
  - Certains autres types évolués spécifiquement dédiés au matériel.
- Les types natifs simulant souvent plus vite que les types évolués SystemC.
- Voici les principaux types de SystemC :
  - Les types bits
  - Les vecteurs de bits
  - Les types entiers
  - Les nombres en virgule fixe
  - Les types temporels

1. Les types de données — Les types bits

SystemC propose deux types de bit : sc_bit et sc_logic

- **sc_bit**:
  - Ce type est maintenant obsolète (depuis SystemC 2.2.0). Il n'est cité ici qu'à titre indicatif. On utilisera plutôt le type **bool**.
  - Ce type est destiné à représenter un bit ne pouvant prendre que deux valeurs, '0' (false) et '1' (true).
  - Il dispose de tous les opérateurs habituels logiques et de comparaison. Il peut aussi être mixé à volonté avec le type bool et les expressions logiques C/C++.
  - Exemple :

```
sc_bit  a;
sc_bit  b;
bool    c;
a = '1';
b = a;
a &= b;
if((i==0) || c || a) {...}
```

1. Les types de données — Les types bits

- **sc_logic:**
  - Ce type est une extension de sc_bit à la logique 4-valuée. Il représente un bit pouvant prendre les valeurs '0', '1', 'Z' et 'X'. Par défaut, un objet sc_logic non initialisé vaut 'X'.
  - Il dispose des mêmes opérateurs que les types sc_bit et bool, et peut être inter-mixé avec eux (le type de l'expression résultante est alors sc_logic).
  - On peut affecter un sc_logic à un sc_bit et vice-versa. L'affectation de 'X' ou 'Z' à un sc_bit provoquera un résultat indéfini et un warning à l'exécution.
  - Exemple:

```
sc_logic out;
bool     data, enable;
if(enable)
  out = data;
else
  out = 'Z';
```


1. Les types de données — Les types entiers

- sc_int<n> et sc_uint<n>: Ces types sont des entiers respectivement signés et non signés, représentés sur n bits, n allant de 1 à 64.
- Ces types sont utiles pour les opérations arithmétiques, mais beaucoup moins rapides à la simulation que le type natif int / unsigned int.
- Ils supportent les opérations arithmétiques habituelles, ainsi que :
  - des méthodes de sélection de bit [], de sélection d'un ensemble de bits .range() et de concaténation ", " / .concat()
  - des méthodes de réduction : .and_reduce(), .or_reduce(), .xnor_reduce(), ...
- Exemple:

```
sc_int<12> val1, val2; // entiers signés sur 12 bits
sc_int<64> res; // entier signé sur 64 bits
sc_int<3> val3; // entier signé sur 3 bits
val1 = 13; // val1 prend "0000000001101"
val2 = val1.range(3,1); // val2 prend "0000000000110", soit +6
val3= val1.range(3,1); // val3 prend "110", soit -2
res = (val1, val2); // res vaut "000000000110100000000000110"
bool c = val3[1]; // c vaut '1' (true)
```

1. Les types de données — Les vecteurs de bits

- sc_bv<n>: Les types vecteurs de bits ne doivent pas être confondus avec les types entiers. Ce sont des tableaux de bits, pas des nombres. Ils ne disposent pas d'opérations arithmétiques.
- C'est un vecteur de bits 2-valués. Il dispose des méthodes de :
  - Sélection de bit [], sélection d'un ensemble de bits .range() et concaténation ","
  - Réduction : .and_reduce(), .or_reduce(), .xnor_reduce(), ...
  - Assignation : =, &=, |=, ^=
  - Manipulation de bits : ~, &, |, ^, <<, >>
  - Egalité : ==, !=
- Exemple:

```
sc_bv<8>   x;
sc_bit     y;
x = "01000111";
y = x[6]; // y vaut '1'
sc_bv<16> data16;
sc_bv<32> data32;
data32.range(15,0) = data16;
data16 = (data32.range(7,0), data32.range(23,16));
(data16.range(3,0),data16.range(15,12)) = data32.range(7,0);
```


1. Les types de données — Les vecteurs de bits

- sc_lv<n>: vecteur de bits 4-valués (sc_logic). Il accepte donc les valeurs 'X' et 'Z' en plus de celles des sc_bv.
- Il est typiquement utilisé pour modéliser des bus de données trois-états (sortie de RAM, ...).
- Son comportement est le même que celui des sc_bv, mais simule moins vite.
- Pour effectuer des opérations arithmétiques sur un sc_bv ou sc_lv, il faut passer par une variable temporaire de type par exemple int ou sc_int => Lors de la conversion, si un 'X' ou 'Z' est présent, le résultat est indéfini, et un warning est déclenché à l'exécution.

1. Les types de données — Les nombres en virgule fixe

- Les systèmes numériques travaillant sur des nombres décimaux les représentent souvent en virgule fixe. SystemC propose quatre types pour représenter ces nombres en virgule fixe, avec différents modèles de quantification et de dépassement.

	Signés	Non-signés
paramètres déterminés à la compilation	**sc_fixed**	**sc_ufixed**
paramètres dynamiques	**sc_fix**	**sc_ufix**

Types flottants en virgule fixe

- Afin d'accélérer les temps de simulation, il est possible de préfixer le nom des classes à l'aide de _fast si vos instances possèdent une mantisse codée sur moins de 33 bits.
- **sc_fixed<>**, **sc_ufixed<>** : ces types ont des paramètres de quantification et de débordement définis à la compilation, et sont respectivement signés et non signés. Syntaxe: sc_fixed<wl, iwl, q_mode, o_mode, n_bits> object_name;
  sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> object_name;

1. Les types de données — Les nombres en virgule fixe

- wl: nombre total de bits dans le mot
- iwl: nombre de bits de la partie entière
- q_mode: mode de quantification (SC_RND, SC_RND_ZERO, SC_RND_MIN_INF, SC_RND_INF, SC_RND_CONV, SC_TRN, SC_TRN_ZERO)
  Détermine le comportement du mot lors d'une opération qui génère plus bits après la virgule qu'il n'y en a de disponibles. Exemple : tronque, arrondi vers zéro, vers moins l'infini, ...
- o_mode: mode de dépassement (SC_SAT, SC_SAT_ZERO, SC_SAT_SYM, SC_WRAP, SC_WRAP_SM)
  Détermine le comportement du mot lors d'une opération qui génère plus de bits avant la virgule qu'il n'y en a de disponibles. Exemple : sature, met à zéro, boucle,..
- n_bits: nombre de bits saturés en cas de dépassement

Exemple:
```
float adder(float a, float b)
{ sc_fixed_fast<4,2,SC_RND,SC_WRAP> Inputa = a;
  sc_fixed_fast<6,3,SC_RND,SC_WRAP> Inputb = b;
  sc_fixed_fast<7,4,SC_RND,SC_WRAP> Output;
 Output = (Inputa + Inputb);
 return (Output); }
```

1. Les types de données — Les nombres en virgule fixe

- **sc_fix<>, sc_ufix<>** : ces types sont les mêmes que sc_fixed et sc_ufixed, mais leurs paramètres peuvent être changés lors de l'exécution.
- Les paramètres sont précisés à l'aide du type sc_fxtypes_params, qui prennent en arguments les mêmes paramètres que sc_fixed et sc_ufixed.
  sc_fxtypes_context précise les paramètres par défaut des sc_fix et sc_ufix.
- Exemple:

```
sc_fxtype_params param1 (10,4,SC_RND, SC_SAT);
// xxxxxx.yyyy
// round to closest representable number
// saturate on overflow
sc_fix_fast valeur (param1);
sc_fxtype_params myparams(SC_RND, SC_SAT);
sc_fxtype_context mycontext(myparams);
sc_fix_fast adder(sc_fix_fast a, sc_fix_fast b)
{ // specify output wl and iwl to be one bit larger
 // than wl and iwl of a sc_fix_fast Output(a.wl() + 1, a.iwl() + 1);
 sc_fix_fast Output(a.wl() + 1, a.iwl() + 1);
 Output = a + b;
 return(Output);}
```


1. Les types de données — Les types temporels

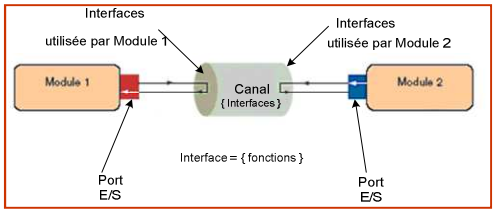
- Une valeur temporelle est de type **sc_time**, et demande deux arguments lors de sa création :
  - un argument de type double spécifiant la valeur,
  - un argument de type énuméré sc_time_unit en spécifiant l'unité.
- sc_time_unit peut valoir :
  - SC_FS : femto secondes
  - SC_PS : picosecondes
  - SC_NS : nanosecondes (unité par défaut)
  - SC_US : microsecondes
  - SC_MS :millisecondes
  - SC_SEC : secondes
- sc_time est muni de méthodes de :
  - d'affectation (=)
  - de comparaison et d'égalité
- Exemple: `sc_time t1( 20, SC_NS );`

2. Les interfaces

- Un port d'un module est connecté à un canal (sauf dans le cas particulier où il est directement connecté au port d'un module parent). Les processus du module communiquent vers l'extérieur au travers de ces ports, en appelant les méthodes du canal auquel il est relié (ce canal est généralement un signal).
- Une interface est un objet C++ ne contenant que les déclarations des méthodes du canal relié à un port, et donc accessibles par l'intérieur du module.
- une interface ne contient pas de code, juste des déclarations de méthodes (purement virtuelles) => tout ce qu'un port a besoin de connaitre c'est le nom des méthodes à appeler, pas leur implémentation
- c'est le canal qui est en charge de fournir l'implémentation des méthodes d'une interface

2. Les interfaces

- Le port est donc un trou dans un module, et l'interface un moyen de se mettre d'accord avec le canal qui y sera branché sur la manière de discuter.
- Dit autrement, quand on accède à un canal depuis l'intérieur d'un module (à travers un port), on ne peut appeler que les méthodes de ce canal déclarées dans l'interface du port.
- Quand on accède à un port, on accède en fait de manière cachée aux méthodes du canal qui lui est relié par l'intermédiaire de l'interface.



- Nous présentons ci après les interfaces standards définies par SystemC, pour les canaux atomiques (signal, fifo, mutex et sémaphore).

2. Les interfaces — sc_signal

- Il existe deux interfaces pour les signaux : sc_signal_in_if pour la lecture seule des signaux (connectés à un port sc_in), et sc_signal_inout_if qui rajoute la notion d'écriture (signaux connectés à un port sc_out ou sc_inout).
- sc_signal_in_if<T> (pour les ports en lecture seule)
  - bool ***event***() : renvoie true si le signal associé a été modifié dans le delta_cycle précédent
  - bool ***negedge***(): renvoie true si le signal a subi un front descendant au précédent cycle de simulation, false sinon.
  - bool ***posedge***(): renvoie true si le signal a subi un front montant au précédent cycle de simulation, false sinon.
  - T& ***read***() : renvoie une référence à la valeur courante du signal associé
  - ***print*** (ostream&) : affiche l'état courant du signal associé sur le flux ostream
  - const sc_event& ***value_changed_event*** () : renvoie une référence à un événement qui sera notifié lors d'un changement d'état du signal associé. Cet événement sera typiquement utilisé dans les fonction wait().

2. Les interfaces — sc_signal

- const sc_event& ***posedge_event***() : renvoie une référence à un événement qui sera notifié lors d'un changement d'état vers true du signal associé (s'il est de type bool ou sc_logic seulement). Cet événement sera typiquement utilisé dans les fonction wait().
- const sc_event& ***negedge_event***() : renvoie une référence à un événement qui sera notifié lors d'un changement d'état vers false du signal associé (s'il est de type bool ou sc_logic seulement). Cet événement sera typiquement utilisé dans les fonction wait().

- sc_signal_inout_if<T> (pour les ports aussi en écriture)
  - toutes les méthodes de sc_signal_in_if
  - void ***write*** (const T&) : permet de changer la valeur du signal associé. Le changement sera effectif au prochain delta_cycle. Un événement ne sera notifié que si la nouvelle valeur diffère de la valeur actuelle.

2. Les interfaces — sc_buffer

- Un buffer se comporte exactement comme un signal, à la différence près qu'un événement de changement d'état est notifié à chaque affectation, même si on lui affecte la même valeur que celle courante. Pour un sc_signal, le changement d'état n'est notifié que si la valeur change.
- Il implémente les mêmes interfaces que sc_signal.

2. Les interfaces — sc_fifo

Comme pour les signaux, les fifo disposent de deux interfaces:

- sc_fifo_in_if <T> (en lecture)
  - T ***read***(), read(T&) : effectue une lecture bloquante de la fifo.
  - bool ***nb_read***(T&) : effectue une lecture non bloquante de la fifo. Renvoie true si une lecture a eu lieu, false sinon.
  - int ***num_available***() : renvoie le nombre d'élément en attente de lecture dans la fifo.
  - sc_event& ***data_written_event***() : renvoie une référence à un événement qui sera notifié lors d'une écriture dans la fifo. Cet événement pourra être utilisé dans une fonction wait().
- sc_fifo_out_if <T> (en écriture)
  - ***write***(T&) : effectue une écriture bloquante dans la fifo
  - bool ***nb_write***(T&) : effectue une écriture non bloquante dans la fifo. Renvoie true si elle été effectuée, false sinon
  - int ***num_free***() : renvoie le nombre d'emplacement libres dans la fifo
  - sc_event& **data_read_event**() : renvoie une référence à un événement qui sera notifié lors d'une lecture de la fifo. Cet événement pourra être utilisé dans une fonction wait().

2. Les interfaces — sc_mutex et sc_semaphore

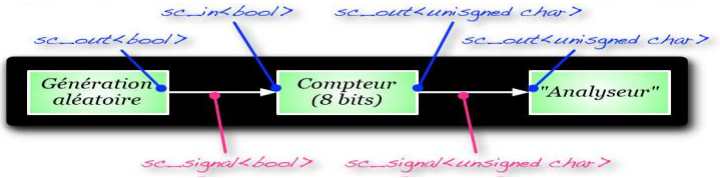
- sc_mutex_if :
  - int ***sc_lock***() : si le mutex n'est pas locké, la fonction le locke, sinon elle se bloque jusqu'à ce qu'il puisse être locké.
  - int ***try_lock***() : si le mutex n'est pas locké, le locke. Renvoie 0 en cas de succès, -1 sinon.
  - int ***unlock***() : délocke un mutex. Les processus en attente de lock sur ce mutex seront alors réveillés.
- sc_semaphore_if :
  - int ***wait***() : acquiert le sémaphore, et décrémente sa valeur de 1. Bloque s'il valait 0.
  - int ***try_wait***() : idem, en non-bloquant.
  - int ***post***() : relâche le sémaphore, et incrémente sa valeur de 1
  - int ***get_value***() : renvoie la valeur actuelle du sémaphore

3. Les canaux de communication

- Les canaux de communication sont les moyens mis à disposition par SystemC pour permettre le transfert de données entre différents modules composant le système.
- On distingue 2 niveaux d'abstraction:
  - Les canaux concrets (des fils, qu'on appelle sc_signal ou sc_buffer),
  - Les canaux abstraits (sc_fifo, sc_mutex, bus, réseau ethernet, pigeon voyageur, etc.).
- Les canaux sont utilisés :
  - à l'intérieur d'un module pour la communication entre deux processus. Les processus peuvent alors accéder directement aux interfaces des canaux.
  - pour la communication entre modules. Les processus accèdent alors au canal par l'intermédiaire d'un port du module. Le port doit alors présenter une des interfaces du canal.

3. Les canaux de communication

- Les canaux implémentent les interfaces. Quand un processus accède à un port, c'est en fait la méthode du canal correspondant qui est exécutée. Cette méthode s'exécute dans le contexte du processus => un processus qui fait une lecture bloquante sur un port connecté à une fifo peut se retrouver bloqué (suspendu), bien que la méthode exécutée soit implémentée par la fifo.
- Les canaux sont connectables uniquement aux ports des composants qui sont compatibles aux interfaces de ces derniers.



- Exemples:
  - ➡ Un fil sur 1 bit sera modélisé par un signal de type sc_signal<bool>.
  - ➡ Un bus de données ou d'adresse sur 32 bits sera un défini comme étant un canal de type sc_signal<sc_uint<32> > ou sc_signal<sc_lv<32> >

3. Les canaux de communication — sc_signal<T>

- Les liens les plus basiques (fils / bus) sont représentés en SystemC à l'aide des "signaux" modélisés dans la classe sc_signal.
- La classe sc_signal<T> est générique permettant au concepteur de ne pas se préoccuper de la taille de ses interconnexions,

  ➡ Lorsqu'on instancie un signal, il faut spécifier le type d'informations transportées.
- Exemples :

  ➡ Si le signal est sur 1 bit (donnée, horloge, ...), la valeur véhiculée entre les composants est un booléen, on utilisera donc : **sc_signal<bool>**

  ➡ Pour un bus d'adresses sur 13 bits, on utilisera: **sc_signal<sc_uint<13> >**

  ➡ Pour un bus sur 32 bits, on pourra utiliser au choix **sc_signal<unsigned int>**, ou **sc_signal<sc_uint<32> >**

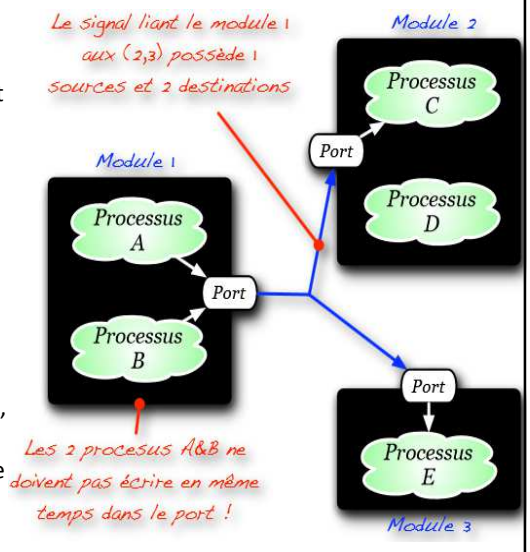
  ➡ Afin de transmettre des données "complexes", on peut créer des signaux transportant des structures : **sc_signal<structure_a_transporter>**

3. Les canaux de communication — sc_signal<T>

- Les signaux sont utilisés :
  - A l'intérieur d'un module pour la communication entre deux processus,
  - A l'extérieur des modules pour permettre la communication entre modules. Dans ce cas, les processus accèdent au signal au travers des ports des modules,
- Remarques :
  - Ne jamais utiliser de variables "globales" pour communiquer entre plusieurs processus cela provoquerait alors des comportements non-déterministes !
  - La seule façon de communiquer des données d'un processus à un autre est en passant par un signal car ces derniers sont gérés par le simulateur.
- Comme en VHDL, la différence entre signaux et variables provient du temps d'affectation des données
  - Variable <=> affectation immédiate,
  - Signal <=> affectation différée, (après que tous les processus aient été exécutés au moins une fois),

3. Les canaux de communication — sc_signal : Règles de construction et d'utilisation

- Règles d'affectation
  - Les signaux ne peuvent être affectés que par un seul processus à la fois. Si un signal est partagé, il faut s'assurer que seul un processus y écrit par "cycle d'horloge" (mutex par exemple),
  - Si un processus écrit plusieurs fois au cours du même cycle dans un signal, seule la dernière écriture sera prise en compte,
- Règles d'interconnexion
  - Un signal ne peut être connecté qu'à un seul et unique port en écriture (producteur de données),
  - Un signal peut être interconnecté à n ports en lecture (consommateurs),



3. Les canaux de communication — sc_signal <T>

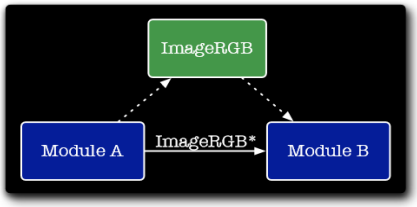
Pour accéder aux données, les méthodes de la classe:

- Lire la valeur contenue dans un signal
  - sc_signal<int> mon_signal;
  - int a = mon_signal;
  - int a = mon_signal.read();
- Affecter une valeur à un signal
  - sc_signal<bool> mon_signal = true;
  - mon_signal.write(false);
- Pour identifier le signal qui a déclenché l'exécution d'un processus (s'il y en a plusieurs dans sa liste de sensibilité par exemple) :
  - bool event( ) : true si le signal a été modifié au cycle précédent, false sinon.
  - bool posedge( ) : true si le signal a subi un front montant au cycle précédent.
  - bool negedge( ) : true si le signal a subi un front descendant au précédent cycle.

3. Les canaux de communication — sc_buffer <T>

- Similaire à sc_signal, à une différence près : sc_signal ne déclenche un événement que si sa nouvelle valeur est différente de la valeur courante, alors que sc_buffer déclenche un événement d'affectation quelque soit la nouvelle valeur.
- Exemple: avec une structure de données complexe

```
struct PixelRGB{
    int rouge;
    int vert;
    int bleu;
};

struct ImageRGB{
    PixelRGB* image;
    int largeur;
    int longueur;
};
```




```
int sc_main (int argc, char * argv []){
    sc_buffer<PixelRGB*> point;
    sc_buffer<ImageRGB*> image;
    return 0;
}
```

3. Les canaux de communication — sc_fifo <T>

- Ce type de canal implémente une fifo (abstraite), de longueur définie lors de son instanciation (paramètre de construction).
- C'est un moyen de communication point à point => il ne peut avoir qu'un seul processus lecteur, et un seul écrivain.
- Cette fifo se comporte comme un signal : une écriture dedans n'est prise en compte qu'à la fin du delta-cycle courant. La valeur écrite ne pourra donc être lue au plus tôt qu'au delta-cycle suivant.
- Elle implémente les interfaces sc_fifo_in_if et sc_fifo_out_if.
- Exemple d'utilisation:

```
// sc_fifo de longueur par défaut égale à 16
sc_fifo<int>  fifo;
int           a,b;
a = fifo.read();
b = fifo;
fifo.write(a);
fifo = b;
if(fifo.num_available() > 3)
a = fifo;
```


3. Les canaux de communication — sc_mutex

- Ce canal implémente un mutex (un verrou d'accès concurrent). Il permet à plusieurs processus d'accéder à une ressource partagée de façon sûre

  => avant chaque accès, un processus devra acquérir le mutex. S'il y arrive, le mutex est verouillé, et seul ce processus qui peut le déverrouiller. Il peut alors accéder tranquillement à la ressource partagée : SystemC garantit qu'aucun autre processus ne pourra acquérir le verrou alors qu'il est verrouillé.
- Si un processus essaye d'acquérir le verrou de façon bloquante (méthode lock()) alors que le verrou est déjà pris, => le processus est suspendu. Il ne sera réveillé qu'à la libération du verrou.
- Les mutex implémentent l'interface sc_mutex_if
- Exemple d'utilisation:

```
sc_mutex   mutex;
... while( true ) {
  wait(clock.posedge_event());
  mutex.lock();
  ... // do something
  mutex.unlock(); }
```

3. Les canaux de communication — sc_semaphore

- Les sémaphores sont une généralisation des mutex à n accès concurrents. Le sémaphore est initialisé à sa construction avec une valeur n.
  - Lorsqu'un processus veut accéder à une ressource partagée, il doit d'abord accéder au sémaphore. Si le sémaphore a une valeur> 0, il est décrémenté et le processus peut accéder à la ressource partagée. Sinon, le processus est mis en veille jusqu'à ce que le sémaphore revienne > 0.
  - Lorsqu'un processus a fini d'accéder à la ressource partagée, il relâche le sémaphore, ce qui a pour effet de l'incrémenter (et éventuellement de réveiller des processus suspendus en attente du sémaphore).
- Si plusieurs processus tentent d'acquérir le sémaphore pendant le même delta_cycle, un seul y arrivera. Mais celui qui y arrivera n'est pas déterminé (car l'ordre d'exécution des processus dans un delta_cycle n'est pas déterminé).

3. Les canaux de communication — sc_semaphore

- Les sémaphores implémentent l'interface sc_semaphore_if
- Exemple d'utilisation :

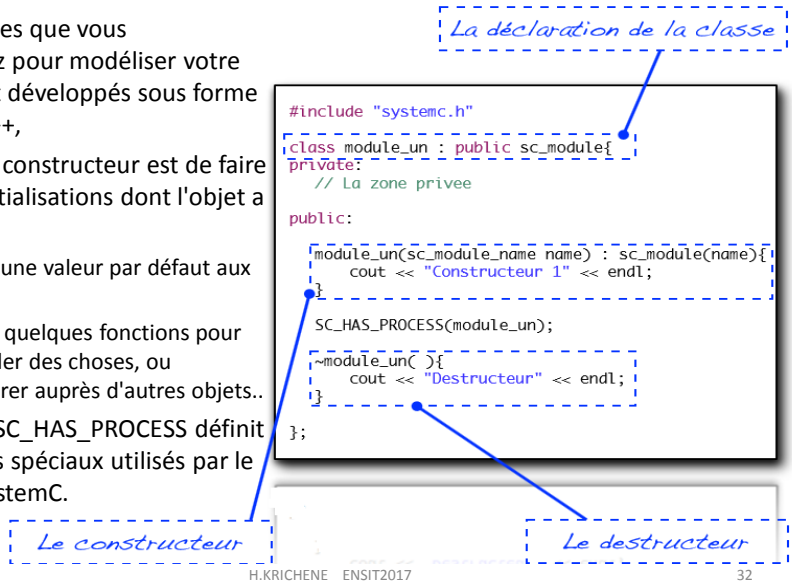
```
// instancie un semaphore à 5 accès concurrents
sc_semaphore sem(5);
// instancie un sémaphore à 1 accès concurrent = mutex
sc_semaphore mut(1);
while( true )
{
 wait(clock.posedge_event());
 sem.wait();
 ... // do something
 sem.post();
}
```

4. Les modules

- Un module en SystemC peut être composé de ports, de canaux de communication ainsi que de processus.
- Exemples :
  - Un processeur est un module (qui contient éventuellement d'autres sous- modules : registres, UALs, ...),
  - Les portes AND, OR, NOT seraient aussi des modules,
  - Une RAM serait aussi un module,
  - Un bus complexe peut aussi être modélisé comme un module,
  - Un réseau (NoC, Internet, etc.) peut être représenté comme un module.

4. Les modules

- Les modules que vous développerez pour modéliser votre système sont développés sous forme de classes C++,
- Le rôle du constructeur est de faire toutes les initialisations dont l'objet a besoin:
  - donner une valeur par défaut aux variables,
  - appeler quelques fonctions pour pré-calculer des choses, ou s'enregistrer auprès d'autres objets..
- La macro SC_HAS_PROCESS définit des symboles spéciaux utilisés par le scheduler SystemC.



4. Les modules

❖ Pour simplifier la vie des concepteurs, un certain nombre de macros ont été définies:

Les constructeurs doivent toujours appeler le constructeur du parent

L'utilisation des macros définies par SystemC permet de simplifier l'écriture du code source des modules

```
#include "systemc.h"

class module_un : public sc_module{
private:
   // La zone privee

public:

   module_un(sc_module_name name) : sc_module(name){
       cout << "Constructeur 1" << endl;
   }

   SC_HAS_PROCESS(module_un);

   ~module_un( ){
       cout << "Destructeur" << endl;
   }

};
```

```
SC_MODULE(module_deux)
{
private:
   // La zone privee

public:
   SC_CTOR(module_deux)
   {
   }

   ~module_deux( ){
       cout << "Destructeur" << endl;
   }
};
```


4. Les modules

N.B. Faites attention en utilisant les macros!!

```
SC_MODULE(module_deux)
{
private:
   // La zone privee

public:
   SC_CTOR(module_deux)
   {
   }

   ~module_deux( ){
       cout << "Destructeur" << endl;
   }
};
```

Ce constructeur correspond à :
module_deux(sc_module_name name)

```
void ma_fonction( ){
   module_deux  m1();
   module_deux  m2("un_nom");
   module_deux *m4 = new module_deux("???");
   delete m4;
}
```

L'instanciation d'un objet en SystemC requiert un nom pour l'instance crée (le simulateur s'en sert pour les messages d'erreur).

⇒Donc vous ne créez jamais d'objets à l'aide du constructeur vide (sans argument) !

4. Les modules

- Exemple d'un module, nommé "mémoire", en écrivant son constructeur à la main: ici on va passer un argument supplémentaire au constructeur : le nom d'un fichier pour initialiser la mémoire.
- N.B. Si on ne passe pas d'argument supplémentaire au constructeur => on peut utiliser la macro SC_CTOR, qui prend en argument le nom du module et qui se charge d'appeler SC_HAS_PROCESS.

```
SC_MODULE(memoire)
{
    // Constructeur
    mon_module (sc_module_name name, char * file_name) : sc_module(name)
    {
        // lecture du fichier, et initialisation de la mémoire
        FILE * fin = fopen(file_name, "r");

        ...
    }

    // Variables, processus, méthodes, canaux, etc..

    // On va instancier des processus, donc macro obligatoire :
    SC_HAS_PROCESS(memoire);

    ...
};
```


5. Les ports

- Les entrées-sorties des modules sont appelées des ports.
  - Les ports sont déclarés dans les classes comme des attributs de type public.
- Les ports sont des éléments doublement typés:
  - Un type spécifie le sens du port : entrée du module (sc_in<T>), sortie du module (sc_out<T>) ou entrée et sortie (sc_inout<T>)
  - Un type va spécifier la nature du port qui va interagir avec l'extérieur (int, char*, sc_int, ma_structure, ...),
- La plupart des ports sont destinés à être connectés à des signaux. Ils possèdent donc les méthodes des interfaces propres aux signaux (read, write, event, ...) => Lors d'un appel à la méthode ***read*** d'un port, tout ce que fait ce port c'est de forwarder cet appel à la méthode read du signal auquel il est relié.

5. Les ports

Règles de connexion des ports:

- A l'extérieur du module, ils sont obligatoirement reliés à un canal ou à un port du module parent. On peut éventuellement relier un port à plusieurs canaux, ce qui est bien pratique quand on fait un module qui modélise un bus.
- A l'intérieur du module, on a le choix :
  - soit on relie le port à un canal et on y accède par l'intermédiaire de ce canal,
  - soit on accède directement à partir des processus aux méthodes de lecture et d'écriture fournies par le port du module. Cette solution est préférable car elle simplifie l'écriture du code et ralenti moins les phases de simulation.
- Un port ne peut pas être connecté à rien : cela produit une erreur à la simulation.

5. Les ports

Spécialisés ou pas ?

- Quand on déclare un port, on a le choix :
  - soit on déclare le port à la main, en le faisant hériter d'une interface spécifique,
  - soit on utilise un type de port spécialisé pour le canal auquel il est destiné à être relié. Cette deuxième solution est souvent la moins verbeuse, et les ports spécialisés fournissent aussi des méthodes additionnelles facilitant la vie.

    => SystemC fournit des ports spécialisés pour les signaux et les fifo.

    => Pour les autres canaux, il faudra prendre la solution manuelle.

5. Les ports: Port Normal

- Avant de faire une déclaration manuelle, il faut connaitre le nom de l'interface du canal qui lui sera connecté.
- Syntaxe : sc_port<interface_type, N> port_name;
  - interface_type est le nom de l'interface
  - N est le nombre maximal de canaux qui pourront être connectés à ce port. 0 signifie "illimité". S'il n'est pas précisé, il vaut 1.
  - port_name est le nom du port
- Pour accéder au canal qui lui est relié, on utilise l'opérateur

  ->
- Si un port est relié à plusieurs canaux, on utilise l'opérateur **[]** pour spécifier auquel on s'adresse.

5. Les ports: Port Normal

- Exemple de déclaration :

```
SC_MODULE(mon_module) {

   //  Port "adresse" :
   //     - relié à un signal de type sc_signal<int> maximum,
   //     - en entrée : interface sc_signal_in_if
   //  Pour ce port, on aurait pu utiliser un type spécialisé
   sc_port<sc_signal_in_if<int> > adresse;


   // Port "data" :
   //    - relié à 2 signaux de type sc_signal<int> maximum,
   //    - bidirectionel : interface sc_signal_inout_if
   // Pour ce port, on aurait pu utiliser un type spécialisé
   sc_port<sc_signal_inout_if<int>,2> data;


   // Port p1
   //    - relié à une fifo qui stocke des char
   //    - port en entrée : interface sc_fifo_out_if
   sc_port<sc_fifo_in_if<char> > p1;


   // Port p2
   //    - relié à une fifo qui stocke des sc_bv<27>
   //    - port en sortie  : interface sc_fifo_in_if
   sc_port<sc_fifo_out_if<sc_bv<27> > > p2;

   ...
};
```

- Exemple d'utilisation de ces ports :

```
char i;
sc_bv<27> j;

data[0]->write(12);

if(data[1]->read() == 14) { ... };

ad = adresse->read();

// lit une valeur dans la première fifo
i = p1->read();

// écrit une valeur dans la deuxième s'il reste de la place
if(p2->num_free() > 0)
    p2->write(j);
```

5. Les ports: Port Spécialisé

- Quand un port est relié à un signal ou une fifo, il existe des types de ports spécialisés, plus simples à instancier :

Type du canal	Entrée	Sortie	Bidir
sc_signal, sc_buffer	sc_in	sc_out	sc_inout
sc_signal_rv	sc_in_rv	sc_out_rv	sc_inout_rv
sc_signal_resolved	sc_in_resolved	sc_out_resolved	sc_inout_resolved
sc_fifo	sc_fifo_in	sc_fifo_out	(pas de sens)

- Exemple : le type sc_in<T> est équivalent à sc_port<sc_signal_in_if<T>, 1> .

  On accède à ces types de port non pas par l'opérateur -> mais directement par l'opérateur "."

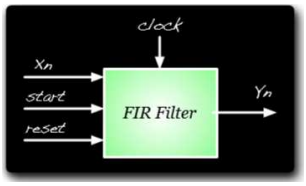
```
SC_MODULE(mon_module) {

  sc_in<bool> clock;
  sc_in<unsigned long> address;
  sc_inout_rv<32> data;

  sc_fifo_in f1;
  sc_fifo_out f2;

  ...

    ad = address; // ou bien ad=address.read() si pb de transtypage
    data.write(0xdeadbeef);

    if(clock.event()) {
      data = "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ";
    }

    i = f1.read();
    if(f2.num_free())
      f2.write(j);
```

41

5. Les ports: Port Spécialisé

- ***sc_signal_rv<int n>***, ***sc_signal_resolved***: ce sont en fait des sc_signal<sc_lv<n>> et sc_signal<sc_logic>, autrement dit des signaux trois-états sur un ou plusieurs bits. Ils sont utilisés pour modéliser des bus 3-états.
- Leurs méthodes sont les mêmes que pour un sc_signal<sc_lv<n> > et sc_signal<sc_logic>.
- Remarque: les anciennes versions de SystemC définissaient trois types spéciaux pour des horloges.
  - typedef sc_in<bool> sc_in_clk
  - typedef sc_out<bool> sc_out_clk
  - typedef sc_inout<bool> sc_inout_clk

5. Les ports: Exemples d'utilisation des ports




```
SC_MODULE(FIR_Filter)
{
    sc_in <sc_uint<32>>  xn;
    sc_out<sc_uint<32>>  yn;
    sc_in <bool> clock;
    sc_in <bool> reset;
    sc_in <bool> start;

    SC_CTOR(FIR_Filter)
    {
    }
};
```

```
SC_MODULE(mon_module)
{
  sc_in_clk            clock;
  sc_in <bool>         reset;
  sc_in <sc_uint<32> > in_A;
  sc_in <sc_uint<32> > in_B;
  sc_out<sc_uint<32> > out;

  ...

  sc_uint<32> r = a.read() + b.read();
  if( reset = '1' )
      out.write( 0 );
  else if( clock.event() ){
      out.write( r );

  ...
}
```

Déclaration des ports

Utilisation des ports pour lire et transmettre des données

5. Les ports: connexion des ports entre modules

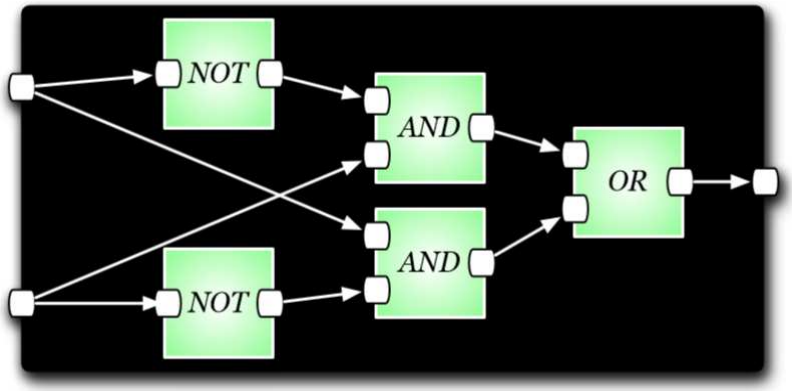
- La connexion des ports aux interfaces se réalise au travers d'un appel de méthodes (celles appartenant aux ports).
  - L'argument passé à la méthode correspond au signal (ou à sa classe dérivée) auquel il est connecté.
- Les connexions sont généralement réalisées dans le constructeur.
- Exemples
  - Exemple d'un composant modélisant une porte XOR à partir de portes logiques de base.

```
// Constructeur
SC_CTOR(compteur) {
    SC_METHOD(compte);
    sensitive_pos(clock);

    // allocation des objets
    current_value = new sc_signal<unsigned char>;
    le_registre = new reg8("Registre");

    // Connexion
    le_registre->in(*current_value);
    le_registre->out(out);
}
```

Attention, les signaux sont déclarés dynamiquement à l'aide de pointeurs car sinon ces objets sont détruit à la sortie du constructeur impliquant un plantage lors de la simulation !!!

5. Les ports: Exemple 1 (porte XOR structurelle)

Structure d'une porte XOR à base de (AND, OR, NOT)



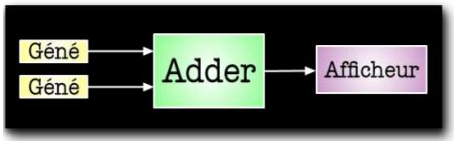
5. Les ports: Exemple 1 (porte XOR structurelle)

```
SC_MODULE(Porte_XOR)
{
 private:
     Porte_NOT *not_1;
     Porte_NOT *not_2;
     Porte_AND *and_1;
     Porte_AND *and_2;
     Porte_OR  *or_1;
     sc_signal<bool> *signal_1;
     sc_signal<bool> *signal_2;
     sc_signal<bool> *signal_3;
     sc_signal<bool> *signal_4;
 public:

   sc_in<bool> a, b;
   sc_out<bool> s;

     SC_CTOR(Porte_XOR){
          // ...................
          // ...................
     }

     ~Porte_XOR(){
          // ...................
          // ...................
     }

};
```

```
not_1 = new Porte_NOT("NOT_1");
not_2 = new Porte_NOT("NOT_2");
and_1 = new Porte_AND("AND_1");
and_2 = new Porte_AND("AND_2");
or_1  = new Porte_OR ("OR_1");
signal_1 = new sc_signal<bool>();
signal_2 = new sc_signal<bool>();
signal_3 = new sc_signal<bool>();
signal_4 = new sc_signal<bool>();
not_1->a( a );
not_1->s( *signal_1 );
not_2->a( b );
not_2->s( *signal_2 );
and_1->a( a );
and_1->b( *signal_2 );
and_1->s( *signal_3 );
and_2->a( *signal_1 );
and_2->b( b );
and_2->s( *signal_4 );
or_1->a( *signal_3 );
or_1->b( *signal_4 );
or_1->s( s );
```

```
delete signal_1;
delete signal_2;
delete signal_3;
delete signal_4;
delete not_1;
delete not_2;
delete and_1;
delete and_2;
delete or_1;
```

5. Les ports: Exemple 2 (Chaîne de calcul)

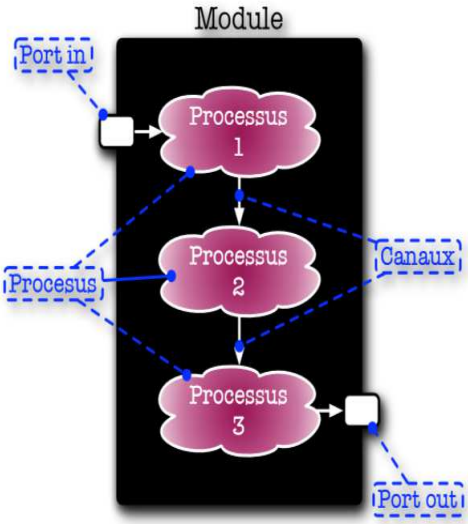


La fonction principale (“main”) du programme doit réaliser l'interconnexion des blocs avant de lancer la simulation du système.

```
#include "Adder.h"
#include "Gene.h"
#include "Terminal.h"

int sc_main (int argc, char * argv [ ]){
    Gene  g1("Data_Generator_1");
    Gene  g2("Data_Generator_2");
    Adder add("Adder_1");
    Terminal term("Terminal_1");

    sc_buffer<int>  in1;
    sc_buffer<int>  in2;
    sc_buffer<int>  out;

    g1.s(in1);   g2.s(in2);
    add.a(in1); add.b(in2); add.s(out);
    term.a(out);

    cout << "Lancement de la simulation..." << endl;
    sc_start(200,SC_NS);
    cout << "Fin de la simulation..." << endl;

    return 0;
}
```


6. Les processus

- Les processus constituent le cœur des modules. Ils sont en charge de la description du comportement du composant,
- Un processus est défini comme étant une méthode,
  - ➔ A la création de l’objet, on va spécifier au simulateur quelles méthodes doivent être exécutées en cas d’apparition de stimulus externes,
- Les processus vont communiquer entre eux à l’aide de canaux de communication.



6. Les processus

- SystemC met à la disposition du concepteur 3 types de processus possédant des propriétés différentes:

  ➡ SC_METHOD : les processus de ce type sont privilégiés pour les composants dont le comportement est déclenché à chaque changement d'une de ses entrées. Ce type est souvent utilisé pour modéliser les composants combinatoires (asynchrones),

  ➡ SC_THREAD : les processus de ce type sont mis en œuvre pour gérer les comportements synchrones ou la liste de sensibilité peut évoluer dans le temps,

  ➡ SC_CTHREAD : les processus de ce type correspondent à un sous-ensemble de SC_THREAD. Ces processus évoluent uniquement sur les fronts d'horloge (sensibilité unique).

- Le choix du type de processus va être réalisé par le concepteur en fonction du comportement à modéliser.

6. Les processus

Un processus est instancié en 4 étapes:

1. Déclaration du processus
   - Les processus sont des méthodes de la classe définissant le comportement temporel du module. Comme toute méthode il doit avoir une déclaration du prototype (.h) et une implémentation du cœur de la méthode (.cpp),
2. Enregistrement
   - Les processus (méthodes) ne sont pas appelés directement, c'est le simulateur SystemC qui les appelle ou réveille lorsqu'il reçoit un signal sur lequel ils sont sensibles.
   - Le simulateur doit tout d'abord savoir quelles méthodes sont des processus ainsi que la catégorie de processus associé.
3. Déclaration de la liste de sensibilité par défaut
   - Le simulateur événementiel a besoin de savoir qu'elles sont les actions qui doivent réveiller chacun des processus déclarés précédemment.
4. Implémentation
   - Comme pour toutes méthodes, il faut fournir une implémentation de la méthode, généralement réalisée dans le fichier (.cpp) associé à la classe.

6. Les processus: **SC_METHOD**

- Les **SC_METHOD** sont des processus particuliers qui permettent de modéliser le comportement de composants dont la liste de sensibilité est figée (statique) lors de l'exécution de la simulation,
- Les SC_METHOD ne sont pas des threads à part entière car ils doivent rendre la main après chaque exécution (return;) sinon cela bloque la simulation !
- Les processus de classe SC_METHOD sont lancés à chacun des changements intervenus sur leur liste de sensibilité (A puis B => 2 exécutions),
- Remarques :
  - Les variables locales contenues dans des méthodes de type SC_METHOD sont ré- initialisées à chaque entrée dans le processus.
  - Si des données doivent être mémorisées, il faut le faire à l'aide d'attributs dans la classe (variables globales dans ce cas de figure).

6. Les processus: **SC_METHOD**

Gestion de la liste de sensibilité du processus:

- La spécification de la liste de sensibilité se fait dans le constructeur du module (après la déclaration du processus et de son type),
  - Pour ajouter un signal dans la liste de sensibilité d'un processus, on se sert de la méthode sensitive( ). Cette méthode prend en argument un signal et s'applique à la dernière SC_METHOD enregistrée.
- Il est possible de filtrer les fronts montants ou descendants (uniquement les signaux sur 1 bit),
  - Déclarer la SC_METHOD comme sensible à tous les fronts du signal (sensitive << clock), et discriminer dans le corps du processus les fronts (if clock.posedge( ) {...}).
  - On peut aussi filtrer les fronts dans la liste de sensibilité à l'aide des constructions suivantes : sensitive << clock.pos( ) et/ou sensitive << mon_signal.neg( ), ce qui accélère les simulations (moins d'exécution des processus).
- Processus avec plusieurs sensibilités
  - sensitive << clock.pos() << clock.neg();
  - sensitive << Input_1 << Input_2;

6. Les processus: **SC_METHOD**

❖ Exemple de définition d'une SC_METHOD:

```
SC_MODULE(mon_module) {

    sc_in<bool> clk;              // Signaux arrivant en entrée de
    sc_in<bool> reset;            // la classe

   void comportement();           // Méthode contenant le comportement

    SC_CTOR(mon_module) {         // Constructueur de la classe
        SC_METHOD(ma_sc_method);  // Enregistrement du processus
        sensitive << enable;      // Insertion des signaux a gérer
        sensitive(reset.pos());   // dans la liste de sensibilité
        // ... ... ... ... ... ...
   }
}
```

On spécifie le type du processus ainsi que les signaux auxquels il est sensible

6. Les processus: **SC_METHOD**

❖ Module implémentant l'addition (Adder):

Adder.h

```
SC_MODULE(Adder)              // module (class) declaration
{
   sc_in<int> a, b;           // ports
   sc_out<int> s;

   void do_add();             // process

   SC_CTOR(Adder)             // constructor
   {
       SC_METHOD(do_add);     // register do_add to kernel
       sensitive << a << b;   // sensitivity list of do_add
   }

};
```

Adder.cpp

```
void Adder::do_add()
{
   s.write( a.read() + b.read() );
   // Identique ^ => s = a + b;
}
```

6. Les processus: **SC_METHOD**

❖ Module d'affichage des données:

Terminal.h

```
SC_MODULE(Terminal)
{
   sc_in<int> a;

   void do_print();

   SC_CTOR(Terminal)
   {
       SC_METHOD(do_print);
       sensitive << a;
   }

};
```

Terminal.cpp

```
#include "Terminal.h"

void Terminal::do_print()
{
   cout << "Time = " << sc_time_stamp();
   cout << " => Valeur recue : " << a << endl;
}
```


6. Les processus: **SC_METHOD**

❖ Module d'affichage des données flexible:

Terminal_T.h

```
template<class T = int>
SC_MODULE(Terminal_T)
{
   sc_in<T> a;

   void do_print();

   SC_CTOR(Terminal_T)
   {
       SC_METHOD(do_print);
       sensitive << a;
   }
};
```

Terminal_T.cpp

```
template<class T>
void Terminal_T<T>::do_print()
{
   cout << "Time = " << sc_time_stamp();
   cout << " => Valeur recue : " << a << endl;
}

Terminal_T<int>    *inst_1;
Terminal_T<float>  *inst_2;
```

6. Les processus: **SC_THREAD**

- Les **SC_THREAD** sont des processus particuliers implémentés sous forme de threads autonomes dans le simulateur.
- Les SC_THREAD permettent de modéliser des processus dont la liste de sensibilité peut évoluer dynamiquement durant l'exécution.
- Les SC_THREAD sont des processus développés autour d'une méthode généralement composée d'une boucle infinie qui s'endort en attente d'événements. Ces processus ne sont plus exécutés à chaque apparition d'un événement, ils sont réveillés !
- Remarques
  - Le SC_THREAD est mis en sommeil (attente d'un nouvel événement) à l'aide de la méthode wait( ).
  - Si un thread s'arrête en arrivant au bout du code à exécuter ou rencontre une instruction "return" alors il ne s'exécutera plus !
  - L'état des variables internes à un processus de type SC_THREAD sont conservées en l'état lors du réveil du processus car il est juste endormi...

6. Les processus: **SC_THREAD**

Mise en œuvre des processus SC_THREAD:

- Création des SC_THREAD
  - Le processus de création des SC_THREAD est exactement identique à celui de création d'une SC_METHOD, en remplaçant cette dernière par la macro SC_THREAD...
  - La déclaration et la gestion de la liste de sensibilité est aussi identique.
- Quand utiliser un SC_THREAD ?
  - Quand on a besoin d'introduire des notions de temps dans les modèles (attendre le déclenchement d'un événement, ou aussi un délai relatif avant d'entreprendre une action),
  - Modéliser des processus séquentiels, typiquement des machines synchrones ou asynchrones à états implicites (décrites étape par étape).
  - Modéliser des processus qui stockent localement de l'information entre deux activations successives,
  - Modéliser des processus accédant à des ressources bloquantes (entrée-sorties, fichiers, Fifo, sémaphores, réseaux, ...),
  - Si on a le choix, les SC_METHOD simulent beaucoup plus rapidement.

6. Les processus: **SC_THREAD**

❖ Différentes mises en attente pour un processus:

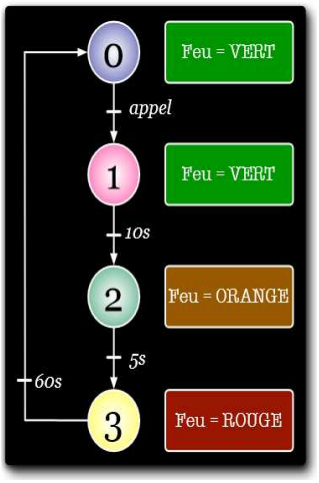
=> Afin de simplifier l'écriture des modèles en SystemC, il est possible de spécifier la mise en attente d'événements de différentes manières:

```
// Process P1 (SC_THREAD, sensitive to clock.pos())
wait();                        // wait for static sensitivity list
wait(e1);                      // wait for event
wait(e1 | e2 | e3);            // wait for first event
wait(e1 & e2 & e3);            // wait for all events
wait(200, SC_NS);              // wait for 200 ns
wait(200, SC_NS, e1 | e2);     // wait, 200 ns timeout
```

Dynamic sensitivy

6. Les processus: **SC_THREAD**

❖ Exemple d'un automate gérant un feu rouge:




```
// Ce processus est un SC_THREAD possédant
// dans sa liste de sensibilité une hologe
// de fréquence 1Hz
void Gestion_Feu::mon_thread() {
    while ( true ) {
        couleur_feu = VERT;
        wait(appel_bouton);

        // Temporisation de 10 secondes
        // puis le feu doit passer ^ l'orange
        for(int i=0; i<10; i++) wait( );
        couleur_feu = ORANGE;

        // Temporisation de 5 secondes
        // puis le feu doit passer au rouge
        wait( 5, SC_S );
        couleur_feu = ROUGE;

        // Temporisation d'une minute
        // puis on repasse au vert (initial)
        wait(60, SC_SC);
    }
}
```

6. Les processus: **SC_THREAD**

❖ Exemple: modélisation d'une horloge 50% et 90%

```
SC_MODULE(Clock_50)
{
    bool state;
    sc_out<bool> out;

    Clock_50(){
        state = false;
    }

    SC_CTOR(Clock_50)
    {
        SC_THREAD(do_gen);
    }

    void do_gen(){
        while( true ){
                state = ! state;
                out.write( state );
                wait(5, SC_NS);
        }
    }
};
```

```
SC_MODULE(Clock_90)
{
   sc_out<bool> out;

   Clock_90();

   SC_CTOR(Clock_90)
   {
       SC_THREAD(do_gen);
   }

   void do_gen(){
       while( true ){
               out.write( true );
               wait(1, SC_NS);
               out.write( false );
               wait(9, SC_NS);
       }
   }
};
```

6. Les processus: **SC_THREAD**

❖ Exemple: Mise en œuvre d'un générateur de données

Gestion relative du temps

```
SC_MODULE(Gene)
{
   sc_out<int> s;

   void do_gen();

   SC_CTOR(Gene)
   {
       SC_THREAD(do_gen);
   }
};
```

```
void Gene::do_gen()
{
   int tmp = 0;
   while( true ){
       tmp = rand()%16;
       s.write( tmp );
       wait( 10, SC_NS );
   }
}
```

6. Les processus: **SC_THREAD**

❖ Exemple: Mise en œuvre d'un générateur de données


```
SC_MODULE(Gene){
   sc_in <bool> clk;
   sc_out<int>  s;

   void do_gen();

   Gene();

   SC_CTOR(Gene)
   {
       SC_METHOD(do_gen);
       sensitive << clk.pos();
   }

};
```

Utilisation d'une horloge externe

```
void Gene::do_gen(){
   s = (int)rand();
}
```


6. Les processus: **SC_CTHREAD**

❖ Les processus déclarés comme étant de nature **SC_CTHREAD** sont proches de leurs homologues définis comme étant des SC_THREAD,

❖ La nuance provient de la liste de sensibilité qui est dans ce cas figée à un unique signal qui jouera le rôle d'une horloge,

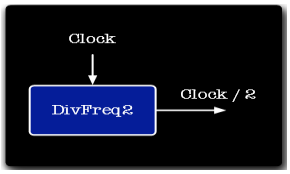
❖ Il est toutefois possible de mixer les mises en sommeil (temporisation) des processus en utilisant des notions de temps (délais)

❖ Remarques:

- ❖ Ce type de processus est majoritairement employé dans les descriptions de bas niveau (Cycle Accurate) ou les composants sont tous synchrones en interne ainsi qu'au niveau de leurs interfaces de communication,
- ❖ Ce sous ensemble des processus a été défini afin de simplifier la tache des outils de CAO qui génèrent automatiquement des descriptions RTL de circuit synchrones,

6. Les processus: **SC_CTHREAD**

❖ Exemple d'utilisation des processus SC_CTHREAD:



On a introduit dans notre modèle SystemC la notion de temps de calcul et/ou propagation dans le processus de calcul.

```
SC_MODULE(DivFreq)
{
   sc_in <bool> clk;
   sc_out<bool> s;

   void do_gen();

   SC_CTOR(DivFreq)
   {
        SC_CTHREAD(do_gen, clk.pos());
   }
};
```

```
#include "DivFreq.h"

void DivFreq::do_gen()
{
    bool state = false;
    while( true ){
        wait( );
        wait(1, SC_NS);
        s.write( ! state );
    }
}
```


7. Le contrôle des simulations

❖ Modélisation du temps en SystemC:

- Il existe une classe nommée sc_time permettant de mesurer le temps,
  - ➢ Utilisation d'un couple (valeur, unité)
- Les unités disponibles sont les suivantes:
  - ➡ SC_SEC : seconde
  - ➡ SC_MS : milliseconde
  - ➡ SC_US : microseconde
  - ➡ SC_NS : nanoseconde
  - ➡ SC_PS : picoseconde
  - ➡ SC_FS : femtoseconde

```
sc_time periode(10, SC_NS);
sc_time delta  (1,  SC_FS);
sc_time latence;

latence = 10 SC_NS; // ERREUR !
```

La définition d'une valeur temporelle n'est possible que lors de l'instanciation d'un objet de type sc_time **uniquement**

7. Le contrôle des simulations

❖ Utilisation de la classe sc_time:

- Le langage SystemC a surchargé aussi un certain nombre d'opérateur de base du langage,
- La classe sc_time supporte ainsi les opérations suivantes :
  - ✓ Affectation,
  - ✓ Addition,
  - ✓ Soustraction,
  - ✓ Multiplication,
  - ✓ Comparaisons (égalité, supérieur, etc.),
  - ✓ Affichage dans un flux standard,

```
sc_time periode(10, SC_NS);
sc_time delta  (1,  SC_FS);

sc_time latence;

sc_time borne_min;
borne_min = periode - delta;

sc_time borne_max;
borne_min = periode + delta;

sc_time delay;
if( periode > latence ){
    delay = latence;
}else{
    delay = 2 * delta;
}
```


7. Le contrôle des simulations

❖ Modélisation d'une horloge à l'aide de la classe sc_clock:

- SystemC donne la possibilité de déclarer des horloges. C'est l'ordonnanceur qui se charge alors de la commutation des signaux.
- Le constructeur accepte les paramètres suivants:
  - ➢ Une chaîne de caractère précisant le nom de l'horloge,
  - ➢ La durée d'une période de l'horloge,
  - ➢ Au besoin le rapport cyclique de l'horloge s'il est différent de 50%.

```
// instancie une horloge de période 10ns
sc_clock clock1("clk100", 10, SC_NS);
sc_time t10(10, SC_NS);
sc_clock clock2("clk100", t10);

// instancie une horloge de période 15.3ns, de rapport cyclique 40%,
// démarrant au temps 45ms et dont le premier état est haut
sc_time tt(45, SC_MS);
sc_clock clock2("clk2", 15.3, SC_NS, 0.4, tt, true);
```

7. Le contrôle des simulations

❖ Modélisation d'une horloge à l'aide de la classe sc_clock:

- En systemC, les horloges offrent les services suivants :
  - period(); // returns sc_time
  - duty _cycle(); // returns a double (fraction of the period)
  - posedge(); // returns a reference to the positive clock edge;
  - negedge(); // negative…
  - read(); // return the current value
  - event(); // detects if there has been a change on clock
  - posedge_event(); // returns an event notification for pos clock edge
  - negedge_event(); // returns an event notification for neg clock edge

7. Le contrôle des simulations

❖ Utilisation du temps (sc_time) dans les processus:

- La modélisation du temps permet l'expression de délais d'exécution dans les processus,

⇒ Etude des performances temporelles & de la synchronisation du système,

- Les spécifications temporelles sont utilisées afin d'exprimer :

⇒ Le temps passé à réaliser les calculs,

⇒ Le temps d'attente avant d'entreprendre une nouvelle action,

- Pas de réelle notion de temps dans les SC_METHOD,

```
void do_gen(){
    while( true ){
            state = ! state;
            out.write( state );
            wait(5, SC_NS);
    }
}
```

Générateur d'horloge

```
void do_FeuRouge(){
   while( true ){
       out.write( FEU_VERT );
       wait(30, SC_S);
       out.write( FEU_ORANGE );
       wait(5, SC_S);
       out.write( FEU_ROUGE );
       wait(60, SC_S);
   }
}
```

Gestion d'un feu rouge

7. Le contrôle des simulations

❖ Lancement de la simulation:

- Afin de vérifier la fonctionnalité et les performances du système, il faut réaliser des simulations fonctionnelles (profiling),
- Le langage inclut des commandes qui permettent d'interagir avec le simulateur afin de le démarrer, régler la durée de simulation et suivre l'état d'avancement.
- Ces dernières sont détaillées ci-dessous :

```
sc_start();                // Simulation infinie

sc_start(200,SC_NS);       // Simulation de 200ns

// Affiche le temps courant du simulateur
cout << "Actual time : " << sc_time_stamp() << endl;
```


7. Le contrôle des simulations

❖ Contrôle de la simulation:

- Si on souhaite une simulation sans fin, il suffit de passer un argument négatif à sc_start(), ou aucun argument.
- Si on passe SC_ZERO_TIME comme durée de simulation, alors la simulation dure un *delta-cycle*.
- La simulation peut être arrêtée n'importe quand depuis un processus, en appelant sc_stop().
- On peut récupérer aussi à tout moment (depuis un processus) :
  - ➢ le temps actuel par la fonction sc_time_stamp(), qui renvoie le temps courant sous forme d'un sc_time
  - ➢ le nombre total de delta-cycles ayant eu lieu depuis le début de la simulation par sc_delta_count(), qui renvoie un entier sur 64 bits
  - ➢ si la simulation est en train de tourner ou non, par sc_is_running().

7. Le contrôle des simulations

❖ Exemple de lancement d'une simulation:

```
int sc_main (int argc, char * argv []){
   Gene  g1("Data_Generator_1");
   Gene  g2("Data_Generator_2");
   Adder add("Adder_1");
   Terminal term("Terminal_1");

   sc_signal<int>  in1;
   sc_signal<int>  in2;
   sc_signal<int>  out;

   g1.s(in1); g2.s(in2);
   add.a(in1); add.b(in2); add.s(out);
   term.a(out);

   sc_start(200,SC_NS);

   return 1;
}
```

Déclaration des composants qui vont être simulés

Déclaration des canaux de communication

Liaison des ports des modules aux canaux de communication

Lancement d'une simulation sur une durée de 200ns

1. Les types de données — Les types définis dans la STL

❖ Le langage SystemC offre aussi accès à toutes les classes et les structures disponibles dans la STL (Standard Template Library):
- Structures de données (vector, queue), des types de données (complex),
- Accès au réseau (socket) aux fichiers (ifstream),
- Algorithmes de tri, de calcul, etc.

❖ Il en va de même avec l'ensemble des autres bibliothèques mathématiques,
- Bibliothèques de calcul sur les matrices,
- Bibliothèques de cryptage / décryptage,
- Bibliothèques de traitement d'images et de traitement du signal,
- Interfaces graphiques + interaction avec l'utilisateur,

❖ Cet interfaçage avec l'ensemble des bibliothèques permet de simplifier les phases de mise au point du modèle fonctionnel au départ.