Introduction à SystemC

Mini-projet

Tarik Graba

2020-2021

Table des matières

1 Historique et concepts

Présentation de l'UE

Objectifs

- Présenter un nouveau *langage* de description du matériel et de modélisation ***SystemC***
- Présenter des concepts de modélisation à différents niveaux d'abstraction
- Regarder sous le capot (SystemC est opensource)

Organisation et évaluation

- Cours suivi de mise en pratique
- Travaux à faire avant la séance suivante
- QCM à la fin

Introduction

Généralités

- SystemC est un "langage" de modélisation et de description du matériel.
  - SystemC est en réalité une bibliothèque `C++`
- SystemC est un standard ouvert (Standard IEEE-1666)
- Une implémentation de référence, libre (open source) est distribuée par Accellera.
- *SystemC permet de décrire du matériel en* `C++`

- Le standard est distribué sous la forme d'un manuel de référence (Language Reference Manual) et est disponible gratuitement sur le site de l'IEEE (Institute of Electrical and Electronics Engineers). Il peut être téléchargé sur cette page du site de l'IEEE.
- Accellera System Initiative est une organisation à but non lucratif dont l'objectif est de développer et de promouvoir des standards industriels pour la conception et la modélisation de systèmes électroniques. La liste des standards soutenus par cette organisation se trouve ici.
- L'implémentation de référence de SystemC est accessible librement (après acceptation de la licence d'utilisation) sur la page dédié du site d'Accellera. Pour ce cours nous utiliserons la version 2.3 du

standard (la version 2.3.2 propose quelques fonctionnalités en avance par rapport à la norme qui sont désactivées par défaut ainsi que la compatibilité avec des compilateurs récents).
- Sur les machines de l'école, la bibliothèque est accessible dans le répertoire `/comelec/softs/opt/systemc/current`. Le répertoire contient le source ainsi qu'une version compilée pour Linux x86_64.

Historique

- 1998, Synopsys "ouvre" son outil Scenic et crée SystemC 0.9
- 2000, Standard OSCI (Open SystemC Initiative)
  - Avec des contributions de Frontier Design et CoWare
- Standard IEEE en 2005 avec la version 2.0
  - SystemC V2.2 correspond au standard IEEE 1666-2005
- Fusion OSCI/Accellera en 2011 et mise à jour du standard
  - SystemC V2.3 correspond au standard IEEE 1666-2011
- Nov. 2016 V2.3.1, passage en licence Apache
- Version actuelle V2.3.3

SystemC plus de 15 ans (bientôt 20) d'évolution !

À la base, c'est des contributions de plusieurs entreprises qui ont été regroupées pour devenir un standard.

La première version de SystemC apportait déjà tous les éléments nécessaires pour la description du matériel :

- Un simulateur évènementiel
- Des processus
- Des signaux

La version 2 de SystemC a apporté des mécanismes de modélisation plus haut niveau dits transactionnels. D'abord sous la forme d'une extension (TLM : *Transaction Level Modeling*) qui a été ensuite intégrée au standard. La version 2.3 de SystemC apporte la 2^e version de cette extension (TLM 2.0).

CoWare et Frontier Design, n'existent plus. Elles ont disparu dans les multiples fusions et rachats entre les différents acteurs du marché de l'EDA.

Pourquoi SystemC

- Pourquoi utiliser `C++` pour décrire du matériel ?
- Pour modéliser *efficacement* un système complet contenant du logiciel et du matériel.

Dans un système sur puce (SoC : *System On Chip*) il y a du matériel :

- des processeurs,
- des accélérateurs (graphiques, audio…),
- des interfaces de communication

et du logiciel exécuté sur ce matériel.

Il faut donc un moyen efficace de modéliser l'ensemble.

Cette modélisation a pour but de :

- Concevoir le système :
  - développer les algorithmes,
  - définir ce qui est fait en logiciel et ce qui est fait en matériel (partitionnement),
- Simuler le système :
  - vérifier les fonctionnalités,
  - avoir un modèle de référence.

C++ a été choisi car :

- c'est un langage objet et on peut s'appuyer sur cela pour représenter des *modules* matériels,
- c'est un langage *efficace* avec lequel on peut obtenir de très bonnes performances,
- c'est un langage que beaucoup de développeurs connaissent déjà.

À quel niveau d'abstraction ?

- Dit autrement, qu'avons-nous besoin de modéliser pour décrire un système numérique complet contenant du logiciel et du matériel ?

Que veut dire *niveau de représentation* ?

- Dans un modèle, avec quel niveau de précision doit-on représenter le comportement ?
- Le temps est-il important ?
  - Oui : On parle de *Timed Model*
  - Non : On parle de *Untimed Model*
- Tous les signaux sont importants ?
  - Oui : On parle de modèle *Bit Accurate*
  - Non : On parle de modèle *Transactionnel*
- A-t-on besoin d'être précis *dans* le modèle ou seulement aux interfaces ?

Pour la fonctionnalité :

Pour décrire un algorithme, on n'a besoin de décrire que la succession d'opérations menant au résultat. Dans cette description *fonctionnelle* le temps nécessaire à l'exécution de ces différentes étapes n'a pas besoin d'apparaître. Ce type de modèle est en général un point de départ pour définir les différentes fonctionnalités que notre système devra réaliser. On parle de modèle *Untimed Functionnal* (UTF).

Dès qu'on veut évaluer les performances d'une implémentation pour connaître, par exemple :

- la puissance de calcul nécessaire (fréquence, nombre d'opérateurs…),
- le nombre d'opérations qu'on peut effectuer dans un temps donné,

on doit introduire une notion de temps.

On doit donc avoir au moins, une description fonctionnelle avec une information de temps sur chaque étape. On parle de modèle *Timed Functionnal* (TF).

Pour l'interface :

Pour décrire des échanges de données dans le système, il peut suffire de les décrire en termes de transactions. Des lectures ou des écritures entre différents éléments du système (un processeur, une mémoire…). S'il y a plusieurs transactions en parallèle, il faut pouvoir garantir l'ordre dans lequel elles ont lieu.

Si en plus, on veut pouvoir vérifier que ces transactions respectent un certain protocole de bus, il faut que le modèle décrive précisément tous les signaux de contrôle.

Que veut dire *niveau de représentation* ?

Dans la conception d'un système sur puce (*SoC*) on passe par plusieurs étapes. À partir d'un modèle fonctionnel, on sépare l'application en partie logicielle et partie matérielle. On parle de *partitionnement*.

Ensuite, les modèles de chaque partie sont *raffinés*.

Pour la partie logicielle, on va utiliser des modèles du support d'exécution de plus en plus précis en :

- utilisant un modèle du système d'exploitation (*OS*),
- faisant apparaître une notion de temps d'exécution,
- en utilisant des plateformes virtuelles ou des simulateurs de jeu d'instruction (ISS : *Instruction Set Simulator*).

Pour la partie matérielle, on va également passer par plusieurs étapes. À partir d'un modèle architectural, dans lequel on définit les différents blocs utilisés, on passe à des modèles transactionnels faisant apparaître les différents transferts de données puis les temps nécessaires pour les traitements et les transferts.

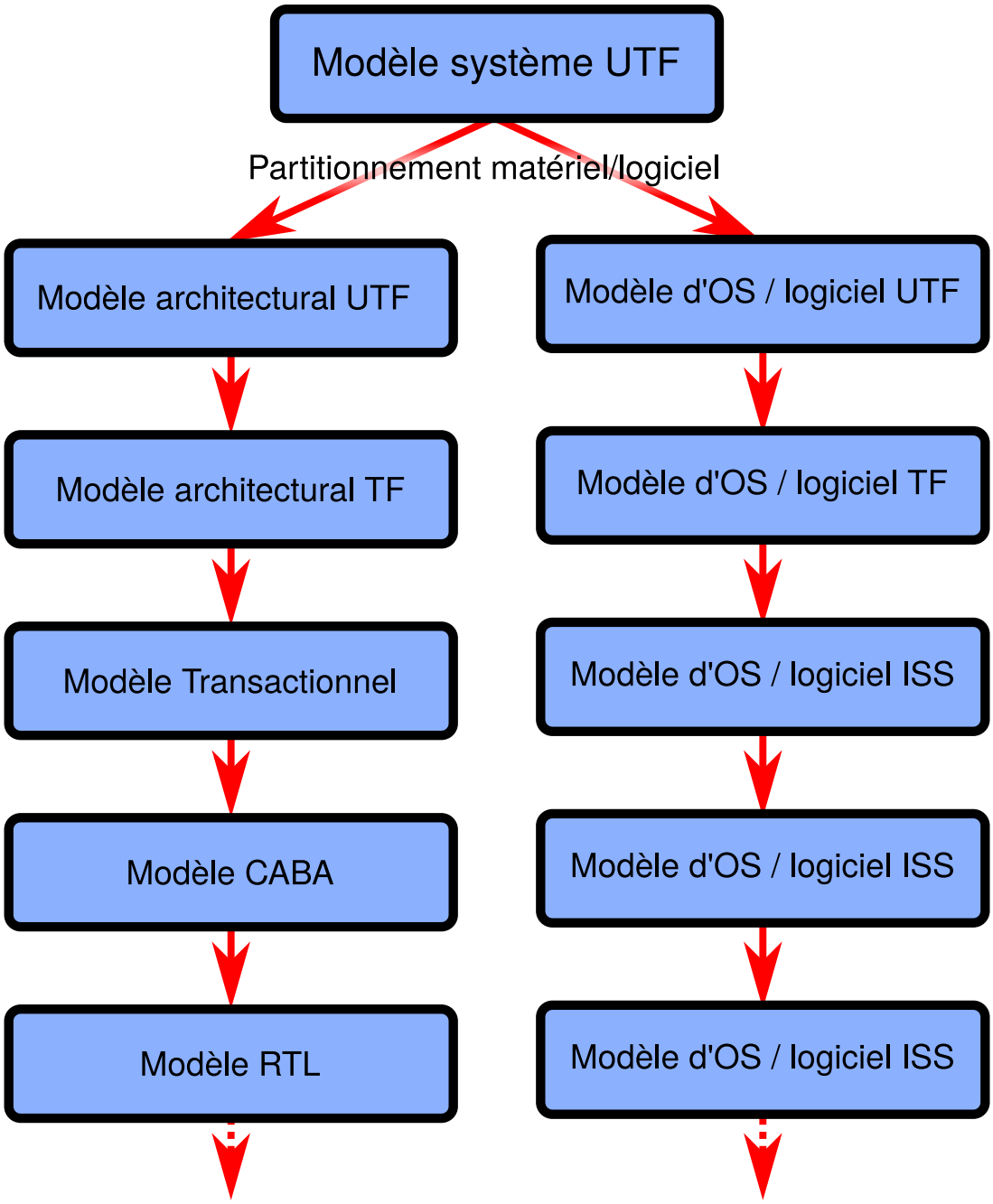

FIG. 1.1: Niveaux d'abstraction dans un flot de conception

Ces modèles deviennent de plus en plus précis au cours du développement jusqu'à obtenir un modèle précis au cycle et au bit près. Ce modèle est dit CABA (*Cycle Accurate Bit Accurate*) qui peut servir de référence à l'écriture d'un modèle RTL synthétisable.

Tout au long de ce *flot de développement* on doit aussi conserver un *modèle de référence* pour vérifier la conformité avec l'application de départ.

Remarque :

Le modèle RTL est un modèle CABA dans lequel, en plus d'être précis au cycle près et au bit près, on doit respecter un style de codage particulier.

SystemC : Un langage pour tous les modèles

L'idée derrière SystemC est d'avoir un langage unique pour écrire des modèles à ces différents niveaux d'abstraction.

Avantages :

- Un seul langage à connaître.
- Faciliter le passage d'un niveau à l'autre et même son automatisation.
- Simplifier en réduisant le nombre de modèles.
- Éviter d'introduire des erreurs en retranscrivant manuellement.

SystemC : Un langage pour tous les modèles

Pourquoi est-ce possible en SystemC ?

- C'est du C++ donc du logiciel.
- On peut utiliser *facilement* des bibliothèques logicielles du système.
- On peut concevoir des modèles qui vont jusqu'au niveau RTL.
- On peut mélanger des modèles de niveaux différents.

SystemC : Un langage pour *presque* tous les modèles

Dans la *réalité* SystemC n'est pas utilisé pour tout faire. Pourquoi ?

Dans l'industrie, SystemC est principalement utilisé comme langage de modélisation. Il est rarement utilisé pour produire des modèles fonctionnels haut niveau ni pour produire des modèles au niveau RTL.

Actuellement, SystemC est utilisé pour la conception de modèles transactionnels permettant de simuler efficacement des systèmes sur puce. Par exemple des modèles de :

- processeurs,
- contrôleurs mémoire,

- interconnexion…

L'utilisation de SystemC permet de concevoir des modèles dont on peut faire évoluer la précision jusqu'à des modèles CABA. Ces modèles peuvent alors être interfacés avec des modules écrits dans d'autres langages de description tels que SystemVerilog ou VHDL.

SystemC : Un langage pour *presque* tous les modèles

Des habitudes

- Il existe des langages spécialisés dans certains domaines (Matlab…) qui facilitent la vie des développeurs.
- Tout le monde ne veut pas apprendre le C++.
- Il a y du code qui existe déjà !

Les outils

- Les outils de synthèse RTL pour SystemC n'ont jamais été développés (ça peut changer)
- Le passage d'un niveau à l'autre n'est pas vraiment automatique.

SystemC

La bibliothèque

SystemC est une bibliothèque logicielle écrite en C++ qui contient les éléments nécessaires à la modélisation d'un système matériel numérique.

C'est-à-dire, ce qu'il faut pour simuler le parallélisme du matériel :

- un moteur de simulation événementiel, des évènements,
- des *signaux*, des types *logiques*,

Plus des extensions pour :

- la modélisation transactionnelle,
- la vérification…

Il faut noter qu'il existe depuis 2010 une extension AMS (*Analog/Mixed-signal*) de SystemC pour la modélisation des systèmes analogiques et mixtes. Cette extension utilise un moteur de simulation analogique (à temps continu) différent du moteur événementiel et des concepts propres à la simulation analogique qui ne seront pas abordés ici.

Structure de la bibliothèque

SystemC est une bibliothèque construite au-dessus de C++ donc tout ce qui est disponible en C++ l'est aussi en SystemC. Ceci est important pour pouvoir réutiliser des bibliothèques logicielles existantes qu'elles aient été écrites en C ou en C++.

Au-dessus de C++, un moteur de simulation événementiel est implémenté. Ce *moteur* permet de gérer les processus et les évènements comme dans d'autres langages HDL (Verilog ou VHDL).

L'implémentation open source de référence distribuée par Accellera contient aussi le moteur de simulation implémenté en C++.

SystemC définit un certain nombre de types utiles à la modélisation du matériel, parmi lesquels :

- des types multi-valué (`0`,1,x,z),
- des bits et des vecteurs,
- des entiers de taille arbitraire,
- et même des nombres en représentation en virgule fixe.

Sont aussi définis les éléments nécessaires à la description hiérarchique : modules, ports et canaux.

En SystemC, les canaux de communication vont au-delà des simples signaux pour permettre des niveaux d'abstraction plus élevés. Un certain nombre de canaux primaires existe dans la bibliothèque (signaux, Fifo…) mais des canaux supplémentaires peuvent être définis si besoin.

Au-dessus de tout ça des bibliothèques supplémentaires sont implémentées. Certaines sont fournies avec l'implémentation de référence telles que les bibliothèques pour la modélisation transactionnelle (TLM/TLM2.0) d'autres sont indépendantes comme la bibliothèque pour la vérification (SVC).

D'autres bibliothèques basées sur SystemC peuvent être fournies par des tiers (vendeur d'outils, d'IP…).

Comment récupérer et installer SystemC

La bibliothèque est disponible sur le site d'Accellera :

http ://accellera.org/downloads/standards/systemc

Peut-être installée sur Linux, Mac OSX et Windows et peut être compilée au moins avec `g++`, `clang++` ou `Visual C++`.

Est déjà installée sur les machines de l'école.

Pour les distributions Linux récente, la bibliothèque est disponible à travers le système de distribution de paquets standard.

Sur Debian (version 10 minimum) ou Ubuntu (19.x minimum) elle peut être installée comme suit :

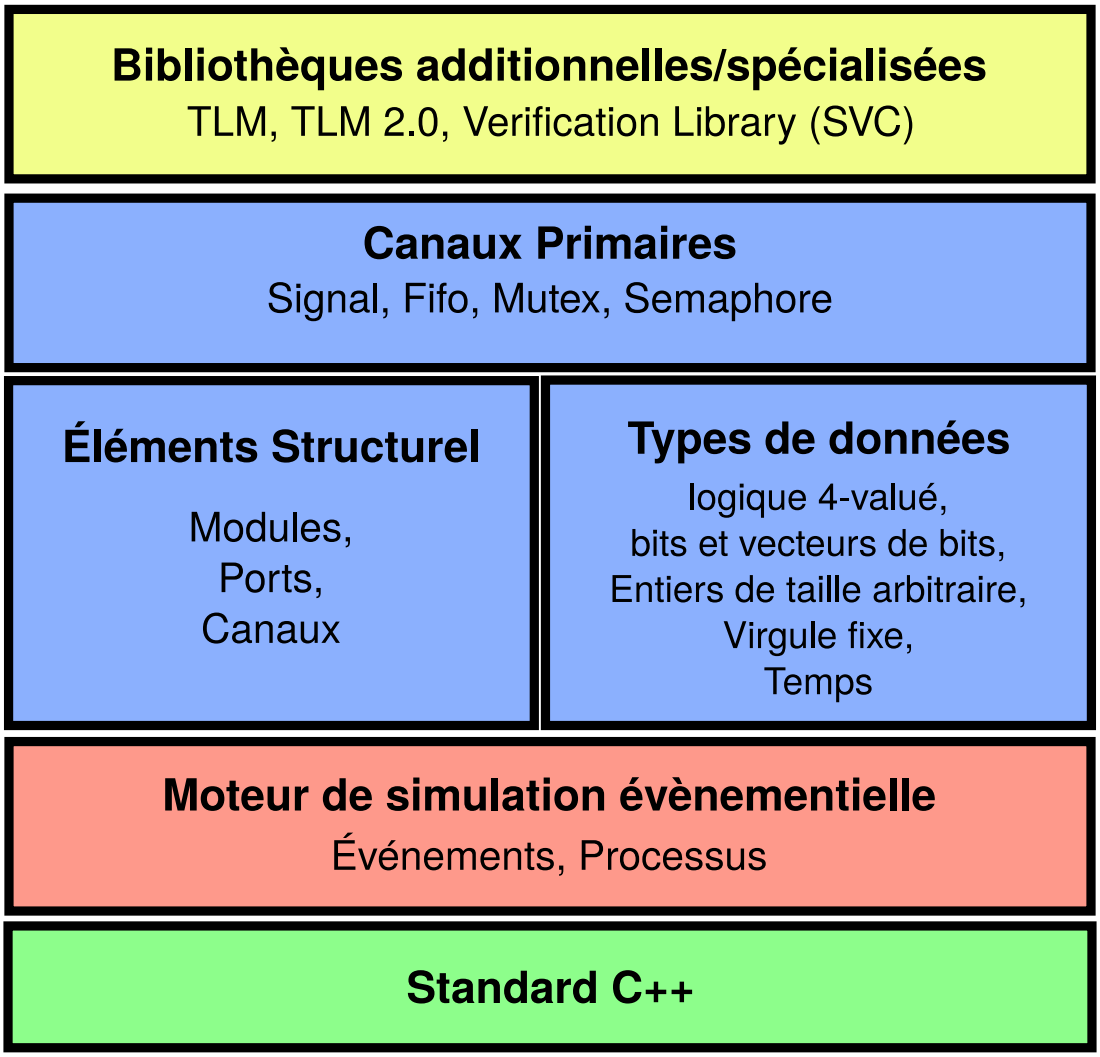

FIG. 1.2: Organisation de la bibliothèque SystemC

```
sudo apt install libsystemc-dev
```

Si vous devez compiler la bibliothèque à partir des sources, il faut télécharger l'archive **Core SystemC Language and Examples** disponible sur le site officiel.

La procédure d'installation est décrite en détail dans le fichier `INSTALL` dans l'archive de la bibliothèque.

Le plus simple sur une machine **Linux** est de l'installer dans `/usr/local/` pour que les fichiers d'en-tête et la bibliothèque soit automatiquement trouvé à la compilation et durant l'exécution de vos programmes.

Pour cela, une fois l'archive décompressée, dans le répertoire que vous obtiendrez, il suffit de faire :

```
./configure --prefix=/usr/local --with-unix-layout
make -j4
sudo make install
sudo ldconfig
```

Le fichier `README` donne la liste des systèmes et compilateurs supportés. Voici un extrait donnant cette liste pour la version 2.3.3 de SystemC.

```
2. This release is supported on the following platform combinations for
   which it has been tested :

    o 64-bit Linux (x86_64)
      (RedHat Enterprise 6 ; SuSE Enterprise Linux 11 ; Debian 9)
      - GNU C++ compiler versions gcc-4.2.2 through gcc-7.2.0
      - Clang C++ compiler versions clang-3.4 through clang-5.0
      - Intel C++ Compiler (ICC 15.0.0)

    o 64-bit Linux (x86_64) with 32-bit compiler (--host=i686-linux-gnu)
      (SuSE Enterprise Linux 11)
      - GNU C++ compiler versions gcc-4.2.2 through gcc-7.2.0
      - Intel C++ Compiler (ICC 15.0.0)

    o 64-bit Linux (aarch64)
      (Ubuntu 16.04)
      - GNU C++ compiler version gcc-4.5.0

    o 64-bit Mac OS X (x86_64)
      (10.12 Sierra)
      - Apple LLVM version 8.0 (clang-800.0.42.1)
      - GNU C++ compiler (MacPorts) versions gcc-4.9.0, gcc-5.4.0, gcc-6.3.0

    o Microsoft Windows
      (Windows Server 2008, Windows 10 Enterprise)
      - Microsoft Visual Studio 2010 (10.0) (Win32 and x64)
      - Microsoft Visual Studio 2013 (12.0) (Win32 and x64)
```

```
      - Microsoft Visual Studio 2015 (14.0) (Win32 and x64)
      - Microsoft Visual Studio 2017 (14.1) (Win32 and x64)

      Warning : The fixed-point datatypes are not working as defined on
                MSVC 2015/2017 (x64) in Release mode.  See RELEASENOTES.

    o Windows Server 2008 (WoW64), Msys 1.0.17(0.48/3/2)
      - MinGW-w64 GNU C++ compiler version 4.9.2 (x86_64)


  This release has not yet been tested or is known not to work as expected
  on the following formerly supported platforms :

    o GNU C++ compiler versions prior to 4.2.x (all platforms)
    o FreeBSD 9.0 or later (x86, x86_64) with GNU C++ compiler
    o HP-UX 11.00 or later with GNU C++ or HP C++ compiler
    o Sun/Oracle Solaris, with GNU C++ or Sun/Solaris Studio compiler
    o Mac OS X prior to 10.12 (Sierra) and/or on the x86, PowerPC architectures
    o Microsoft Visual C++ versions prior to 10.0 (2010)
    o Cygwin 1.7 or later (x86, x86_64) with GNU C++ compiler
    o Msys/MinGW32 GNU C++ compiler
```

Sauf si vous avez un compilateur C++ extraterrestre, vous devriez pouvoir utiliser SystemC.

Aussi, sachez que la version en cours de développement est accessible publiquement sur GitHub.

https ://github.com/accellera-official/systemc

Premier exemple *Hello World*

hello.cpp

```
#include <systemc.h>

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

   cout « "hello world" « endl;

   return 0;
}
```

La fonction principale n'est pas `main` mais `sc_main`.

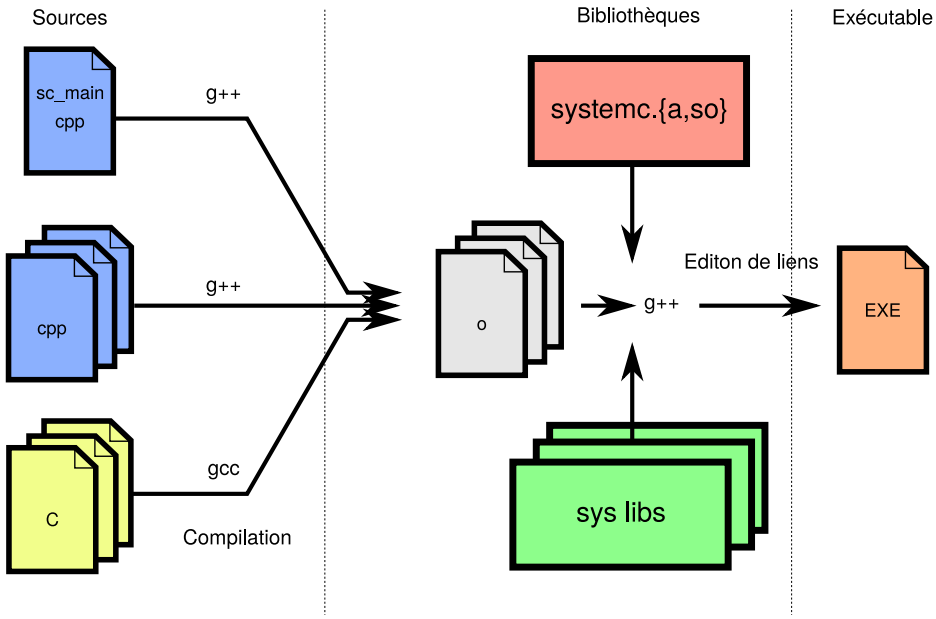

FIG. 1.3: Flot de compilation

La fonction `main` existe quand même, elle est fournie par la bibliothèque SystemC. Elle s'occupe de configurer l'environnement de simulation avant d'appeler la fonction `sc_main` en lui passant les arguments qu'elle a reçus.

Le fichier d'en-tête `systemc.h` contient les définitions des éléments de la bibliothèque. En plus,

- il inclut d'autres fichiers d'en-tête de la bibliothèque standard C++, par exemple `iostream`
- et permet d'utiliser directement certains espaces de noms (`namespace`) relatifs à SystemC et à la bibliothèque standard (`std`).

Si on a besoin de gérer précisément les en-têtes et les espace de noms on peut préférer inclure `systemc`.

Que doit-on compiler ?

Les sources C++ et C sont compilés puis l'édition de liens est faite pour générer un exécutable. Parmi ces sources, doit figurer la définition de la fonction `sc_main`.

La bibliothèque SystemC doit être fournie pour l'édition de liens. Elle apporte les éléments suivants :

- la fonction `main`,
- le simulateur événementiel,
- les autres objets de la bibliothèque SystemC.

Comme, il s'agit d'un flot de compilation standard pour du C++, on peut faire appel à d'autres bibliothèques du système.

Première compilation

`Makefile`

```
SYSTEMC  ?= /comelec/softs/opt/systemc/current
ARCH      = linux64

CPPFLAGS  =  -isystem $(SYSTEMC)/include
CXXFLAGS  =  -Wall -g
LDFLAGS   =  -L$(SYSTEMC)/lib-$(ARCH)
LDLIBS    =  -lsystemc
```

Dans ce `Makefile` minimaliste pour `g++` :

- On précise les chemins de recherche pour les fichiers d'en-tête et les bibliothèques précompilées.
- On ajoute la bibliothèque `systemc` au moment de l'édition de liens.

L'exécution donne :

```
SystemC 2.3.2-Accellera --- Dec 14 2017 09:24:39
Copyright (c) 1996-2017 by all Contributors,
ALL RIGHTS RESERVED

hello world
```

Le message indiquant la version de SystemC est issu de la bibliothèque et montre bien que des choses sont faites avant l'appel à `sc_main`.

---

2 Les types

Le type logique

Le type logique

En C++ le type `bool` permet de représenter une valeur logique

- (`true`, `false`).

SystemC définit, en plus, le type `sc_logic` pouvant prendre 4 valeurs

- (`0`, `1`, `Z`, `X`).

```
sc_logic u;
```

- Il peut être initialisé à partir d'un booléen, un entier ou un caractère.
- Il existe des fonctions de conversion explicite vers un booléen, un entier ou un caractère.
- Il peut être utilisé avec les opérateurs binaires (`~`, `&`, `|`, `^`) et les opérateurs d'égalité.
  - On peut même mélanger un `sc_logic` et un booléen, un entier ou un caractère.

La description complète de ce type se trouve dans la section 7.9.2 de la norme.

Les quatre valeurs correspondent aux états logiques suivant :

- `0` : faux
- `1` : vrai
- `Z` : haute impédance
- `X` : Non défini, inconnu, conflit

En interne, un `sc_logic` contient un élément du type énuméré `sc_logic_value_t` défini comme :

```
enum sc_logic_value_t {
   Log_0=0;
   Log_1,
   Log_Z,
   Log_X
};
```

Donc, ces 4 valeurs possibles sont des valeurs entières.

En plus du constructeur par défaut et du constructeur de copie, un `sc_logic` peut être construit à partir :

- d'un booléen
  - `false` -> `0` et `true`-> 1
- d'un entier
  - seules les valeurs entières entre 0 et 3 sont autorisées
- d'un caractère
  - les caractères `'0'`, `'1'`, `'x'`, `'z'`, `'X'`, `'Z'` peuvent alors être utilisés
  - tous les autres sont équivalents à `x`

Ces constructeurs sont définis comme `explicit` et ne peuvent donc pas être utilisés pour les conversions implicites.

Par contre, les opérateurs binaires ainsi que les opérateurs d'affectation sont définis pour pouvoir utiliser les `sc_logic` avec des booléens, des caractères ou des entiers.

Il existe aussi des méthodes pour convertir ou tester ces valeurs (`to_bool`, `to_char`, `is_01`) qui peuvent être appelées si besoin.

Pour imprimer la valeur d'une variable de type `sc_logic` l'opérateur de flux « a été surchargé. En général, l'opérateur de flux est surchargé pour tous les types définis dans SystemC.

Travail à faire : mise en pratique des `sc_logic`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de type `sc_logic`,
- testez différentes initialisations,
- testez différents opérateurs binaires entre `sc_logic`,
- testez différents opérateurs binaires entre des `sc_logic` et des `bool`,
- Imprimez systématiquement les résultats.

Remarque Il existe aussi un type `sc_bit` ne pouvant prendre que les 2 valeurs (`0`, 1) mais il est déprécié depuis la version 2.2 ce SystemC. Dans ce cas, on lui préférera le type `bool` de C++.

Les vecteurs

Les bit vectors `sc_bv<N>`

Vecteur d'éléments binaires (`bool`).

```
sc_bv<8> x;
```

Le paramètre de template est un entier qui permet de définir la taille du vecteur.

Les logic vectors `sc_lv<N>`

Vecteur de variables logiques (`sc_logic`).

```
sc_lv<8> x;
```

Le paramètre de template est un entier qui permet de définir la taille du vecteur.

Les types `sc_lv` et `sc_bv` sont des vecteurs. Ils ont été conçus pour permettre l'accès efficace à leurs différents éléments. Par contre, ils ne permettent de faire des opérations arithmétiques.

Pour l'arithmétique, nous verrons par la suite des types plus adaptés.

Les vecteurs peuvent être comparés et initialisés à partir d'une chaîne de caractère. Par exemple :

```
// vecteur logic de 4 bits de large
sc_lv<4> x = "01xz";

if (x == "0000")
   ...
```

Ici chaque caractère représente un des éléments logiques.

On peut aussi initialiser un vecteur à partir d'un `bool` ou d'un `char`. Dans ce cas tous les éléments auront la même valeur. Par exemple :

```
// initialise tous les éléments d'un vecteur de 8bits à false?
sc_bv<8> x = false;
sc_bv<8> y = sc_bv<8>(false);
sc_bv<8> z(false);
   ...
```

Les vecteurs peuvent aussi être initialisés à partir d'un entier. Par exemple :

```
//
sc_bv<8> X;

X = 44;
// Imprimera 44
cout « "x---> " « X  « "(" « X.to_uint() « ")" « endl;

// Imprimera 44 aussi car X est sur 8 bits
X = 300;
cout « "x---> " « X  « "(" « X.to_uint() « ")" « endl;
   ...
```

Les opérateurs binaires (~, &, |, ^) ainsi que les opérateurs de décalage (», «) sont supportés.

De plus, certains opérateurs ont été surchargés pour permettre la sélection d'un ou plusieurs éléments. Ils peuvent être utilisés des deux cotés d'une affectation.

```
sc_bv<8> X;

X[0] = false;
```

```
sc_bit b = X[1];

X(7,4) = "1010";
X(3,0) = X(7,4);
```

La méthode `range` peut aussi être utilisée pour la sélection de plusieurs éléments (i.e. `X(n1,n2)` est équivalent à `X.range(n1,n2)`).

La concaténation de deux vecteurs se fait en les mettant entre parenthèses. C'est l'opérateur `','` qui est surchargé.

```
sc_bv<8> X = "11110000";
sc_bv<4> a;
sc_bv<4> b;

(a,b) = X;
X = (b,a);
```

La fonction `concat` peut aussi être utilisée pour la concaténation de deux éléments (i.e. `(n1,n2)` est équivalent à `concat(n1,n2)`).

D'autres méthodes sont implémentées, par exemple :

- Les méthodes de réduction :
  - `and_reduce()`
  - `or_reduce()`
  - `xor_reduce()`
  - `nand_reduce()`
  - `nor_reduce()`
  - `xnor_reduce()`
- Les méthodes de conversion :
  - `to_int()`
  - `to_uint()`
  - `to_long()`
  - `to_ulong()`
  - `to_int64()`
  - `to_uint64()`

Travail à faire : mise en pratique des `sc_lv` ***et*** `sc_bv`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de ces deux types,
- testez différentes initialisations,
- testez différents opérateurs binaires,
- testez différentes conversions entre les deux types,
- imprimez systématiquement les résultats.

Les entiers de taille arbitraire

Remarque

Pour des raisons d'efficacité, les types `sc_lv` et `sc_bv` ne permettent pas de faire d'opérations arithmétiques.

Pour cela, la bibliothèque SystemC fournit des types permettant de manipuler des entiers de taille arbitraire.

`sc_int<N>` et `sc_uint<N>`

SystemC définit ces deux classes pour les entiers de taille inférieure ou égale à 64 bits.

```
sc_int<10> x; // entier signé sur 10 bits
sc_uint<7> y; // entier non signé sur 7 bits
```

En interne, ils utilisent les entiers 64 bits natifs de C++ pour avoir une implémentation efficace des opérateurs arithmétiques. Les opérations de troncature et d'extension sont gérées automatiquement.

Ces types supportent tous les opérateurs arithmétiques et logiques supportés par les entiers en C++. En plus, comme pour les types vecteurs, ils supportent les opérateurs de sélection de bits ou de plages de bits, de concaténation, ainsi que les opérateurs de réduction.

Ils peuvent être initialisés par :

- un entier,
- un `sc_lv` ou un `sc_bv` ce qui est utile pour pouvoir faire des opérations arithmétiques sur ces types.

En pratique, il faut privilégier les types C++ natifs. Si l'on a besoin d'une taille arbitraire qui ne correspond pas à un type natif, on utilise alors les `sc_int` et `sc_uint`. Ce n'est que si l'on a besoin de modéliser un bus 3 états (Z) qu'on utilise les `sc_lv`.

Travail à faire : mise en pratique des `sc_int` ***et*** `sc_uint`

Dans une fonction `sc_main` :
- déclarez plusieurs variables de ces deux types,
- testez différentes initialisations,
- testez différents opérateurs arithmétiques et logiques,
- testez différents conversions entre ces types et des `sc_lv`,
- imprimez systématiquement les résultats.

`sc_bigint<N>` et `sc_biguint<N>`

Si l'on a besoin de représenter des entiers de taille arbitraire supérieure à 64 bits, il faut utiliser ces types.

Ils sont prévus pour permettre des calculs sur une taille illimitée mais pour des raisons pratiques, l'implémentation de référence limite leur taille à SC_MAX_NBITS. Cette valeur est définie dans le fichier kernel/sc_constants.h et peut être modifiée si nécessaire.

Les nombre en virgule fixe

Les types définis

Pour représenter un nombre en virgule fixe on doit préciser sa taille et la position de la virgule.

SystemC propose deux familles de types pour représenter ces nombres :

- Les sc_fixed et sc_ufixed pour lesquels les paramètres sont statiques (définis par des templates)
- Les sc_fix et sc_ufix pour lesquels les paramètres peuvent être définis dynamiquement

---

En plus, des versions _fast sont définies. Ces types utilisent en interne les doubles natifs de C++ pour être plus efficaces mais sont limités à la taille de ceux-ci.

Les sc_fixed et sc_ufixed

```
sc_fixed <wl, iwl, q_mode, o_mode, n_bits> X;
sc_ufixed<wl, iwl, q_mode, o_mode, n_bits> Y;
```

avec :

- wl la taille totale
- iwl la taille de la partie entière
- q_mode le mode de quantification
- o_mode le mode de saturation
- n_bits le nombre de bits qui saturent

Les paramètres wl etiwl doivent être définis à l'instanciaton. Les autres paramètres ont par défaut les valeurs suivantes :

- q_mode : SC_TRN // troncature vers le bas
- o_mode : SC_WRAP // modulo
- n_bits : 0 // aucun bit ne sature

La définition exacte de ces paramètres et des comportements associés se trouvent dans la section 7.10 du standard.

Remarque

Par défaut, les types permettant de représenter des nombres en virgule fixe, ne sont pas disponibles. Pour les utiliser il faut définir la macro `SC_INCLUDE_FX` avant d'inclure `systemc.h`.

Comment les utiliser

Ces types peuvent être initialisés à partir de `float` ou de `double`.

Toutes les opérations arithmétiques et logique sont faites en fonction des paramètres sur lesquels ils ont été définis.

Ces types peuvent, par exemple, être utilisés pour tester simplement une implémentation en virgule fixe et comparer la précision à une implémentation flottante.

Par exemple :

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

// 1 bit pour la partie entière et 2 bits après la virgule
#define WL  3
#define IWL 1

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

   double d1 = 0.1;
   double d2 = 0.9;
   double ds = d1 + d2;

   sc_fixed<WL,IWL> f1(d1);
   sc_fixed<WL,IWL> f2 = d2;
   sc_fixed<WL,IWL> fs = f1 + f2;

   double e = ds - double(fs);

   cout
      « "Double " « ds  « endl
      « "Fixed  " « fs  « endl
      « "Error  " « e   « endl;
   return 0;
}
```

Les sc_fix et sc_ufix

Ces deux types diffèrent des précédents par le fait que les paramètres sont modifiables dynamiquement en fonction d'un contexte. Comme les paramètres sont dynamiques, ils sont plus lents à l'exécution.

Pour définir un contexte :

```
// ici seule la taille est définie, mais tout est modifiable
sc_fxtype_params param(10,3);
sc_fxtype_context ctxt(param);

sc_fix a;   // paramètre définis dans le contexte
```

Exemple d'utilisation :

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

double compare (const double d1, const double d2)
{
   double ds = d1 + d2;
   sc_fix f1(d1);
   sc_fix f2(d2);
   sc_fix fs = f1 + f2;

   return ds - fs;
}

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

   double d1 = 0.1;
   double d2 = 0.9;

   // 1 bit pour la partie entière et 2 bits après la virgule
   sc_fxtype_params param1(3,1);
   // 1 bit pour la partie entière et 4 bits après la virgule
   sc_fxtype_params param2(5,1);
   // 1 bit pour la partie entière et 6 bits après la virgule
   sc_fxtype_params param3(7,1);

   sc_fxtype_context ctxt1(param1);
   cout
      « "The error with " « ctxt1.default_value()
```

```
      « " format is " «  compare(d1,d2)  « endl;

   sc_fxtype_context ctxt2(param2);
   cout
      « "The error with " « ctxt1.default_value()
      « " format is " «  compare(d1,d2)  « endl;

   sc_fxtype_context ctxt3(param3);
   cout
      « "The error with " « ctxt1.default_value()
      « " format is " «  compare(d1,d2)  « endl;

   return 0;
}
```

De façon analogue aux `sc_fixed`, le type `sc_fxtype_params` prend, entre autres, comme paramètres de construction la taille totale de la représentation ainsi que la taille de la partie entière.

Travail à faire : mise en pratique

1. Testez les exemples fournis
2. Écrire une fonction qui permet de trouver le nombre de bits minimum pour représenter **π.sin(π/256 n)** avec une précision de 1/100, 1/1000 et 1/10000 par rapport à un calcul en double précision (**n** étant un entier).

Le temps

Le type `sc_time`

SystemC définit un type pour représenter le temps de simulation ou des intervalles de temps.

Un `sc_time` est construit à partir d'un `double` et d'une unité. L'unité est une des valeurs du type énuméré `sc_time_unit`

```
   sc_time periode(17.22, SC_NS);
```

Le type `sc_time_unit` est défini comme suit :

```
enum sc_time_unit {
   SC_FS = 0,  // femtosecondes
   SC_PS,      // picosecondes
   SC_NS,      // nanosecondes
   SC_US,      // microsecondes
   SC_MS,      // millisecondes
```

```
   SC_SEC      // secondes
};
```

En interne, le temps est stocké sous la forme d'un entier 64 bits qui est un multiple de la résolution temporelle. Cette valeur interne peut être récupérée en utilisant la méthode `value`.

La résolution temporelle est globale à une simulation. Par défaut elle est égale à **1 picoseconde** mais peut être modifiée en utilisant la fonction `sc_set_time_resolution`.

Quelle que soit la résolution, la constante `SC_ZERO_TIME` représente toujours un temps nul.

Le type `sc_time` supporte des opérateurs de comparaison ainsi que certains opérateurs arithmétiques. On peut ainsi additionner deux temps ou les multiplier par un double.

Exemple d'utilisation :

```
#include <systemc.h>

int sc_main (int argc, char * argv[])
{
   sc_time t1(3.51, SC_NS);
   cout « "---> " « t1.value() « ", " « t1 « endl;

   sc_time t2 = 2*t1;
   cout « "---> " « t2.value() « ", " « t2 « endl;

   cout « "La durée t" ;
   if (t1>t2)
      cout « "t1";
   else
      cout « "t2";
   cout « " est plus grande" « endl;

   cout « "---> " « SC_ZERO_TIME.value() « ", " « SC_ZERO_TIME « endl;

   return 0;
}
```

---

3 Simulation et signaux

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.

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`

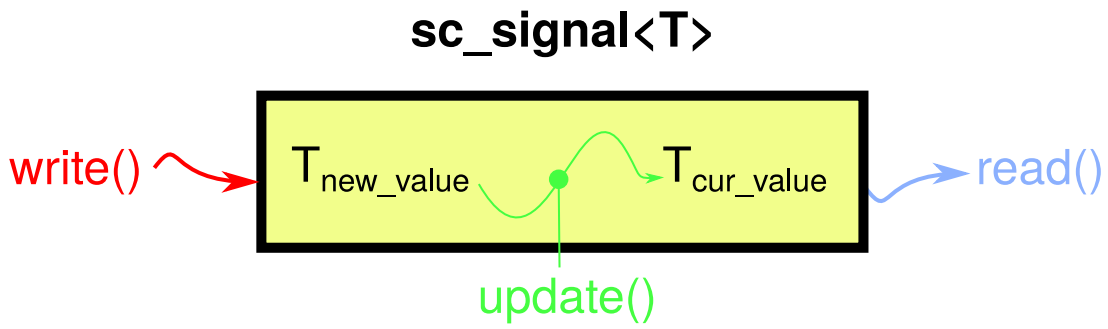

FIG. 3.1: Un sc_signal

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
- 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

---

4 Hiérarchie et modules

Les modules

sc_module

Les modules sont les éléments principaux pour construire une description hiérarchique en SystemC.

Un module SystemC est une classe qui hérite de la classe `sc_module`

```
struct mon_module : sc_module {
 ...
};
```

Un module ne peut être déclaré après le début de la simulation.

Un module SystemC est une classe C++ dans laquelle tous les champs sont publics par défaut.

Il peut donc avoir des attributs et des méthodes comme toute classe C++. Les attributs peuvent bien sur être tout objet C++ ou SystemC, même un autre module.

De plus, on peut séparer la déclaration de la classe de la définition des méthodes dans deux fichiers (`.h` et `.cpp`).

rappel

En C++ :

```
struct X {
    ...
};
```

est équivalent à :

```
class X {
public:
    ...
};
```

Le constructeur d'un module

Un module doit avoir un nom à l'instanciation. Ce nom est passé comme argument de son constructeur et doit être transmis à la classe parent `sc_module`.

Par exemple :

```
mon_module(sc_module_name n):sc_module(n){ ... }
```

La méthode `name()` permet de récupérer ce nom durant l'exécution de la simulation.

La classe `sc_module_name` est une classe intermédiaire qui est utilisée pour définir un nom de module à partir d'une chaine de caractères. À l'usage elle est transparente.

Les macros prédéfinies

Deux macros sont définies pour simplifier cette déclaration :

- `SC_MODULE()` pour déclarer le module
- `SC_CTOR()` pour définir le constructeur

`SC_MODULE(mon_module)` est équivalent à `struct mon_module : sc_module`.

`SC_CTOR(mon_module)` est équivalent à `mon_module(sc_module_name n) :sc_module(n)`

On peut donc déclarant un module en écrivant :

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

   SC_CTOR(mon_module)
   {
   // Initialisation
   }
};
```

Remarque : On verra par la suite que `SC_CTOR` ajoute une information supplémentaire.

Constructeurs additionnels

Si un constructeur additionnel est ajouté il doit aussi appeler le constructeur de la classe parent `sc_module`

Par exemple :

```
mon_module(int i, sc_module_name n):sc_module(n)
{
  ...
}
```

Dans ce cas, la macro SC_CTOR ne peut plus être utilisée.

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

Exemple d'utilisation de SC_HAS_PROCESS

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

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

   SC_HAS_PROCESS(mon_module);
};
```

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

Les ports

sc_in, sc_out, sc_inout

SystemC définit trois types de ports sous la forme de templates :

- sc_in<T> les entrées de type T
- sc_out<T> les sorties de type T
- sc_inout<T> les ports bidirectionnels de type T

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

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

À la construction, on peut donner un nom au port. Ce nom, est utile pour identifier certains problèmes à la simulation.

Si un nom n'est pas donné, un nom générique sera créé. Dans tous les cas, ce nom sera hiérarchique et dépendra du nom du module.

Exemple :

```
#include <systemc.h>

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

   SC_CTOR(foo):clk("clk"), data("data")
   {
      cout
         « "module : " « name()      « endl
         « "clk    : " « clk.name()  « endl
         « "rst    : " « rst.name()  « endl
         « "data   : " « data.name() « endl
         ;
   }
};

int sc_main(int argc, char * argv[])
{
   foo bar("foobar");
   return 0;
}
```

Ou en utilisant les constructions de C++11 :

```
#include <systemc.h>

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

   SC_CTOR(foo)
   {
      cout
```

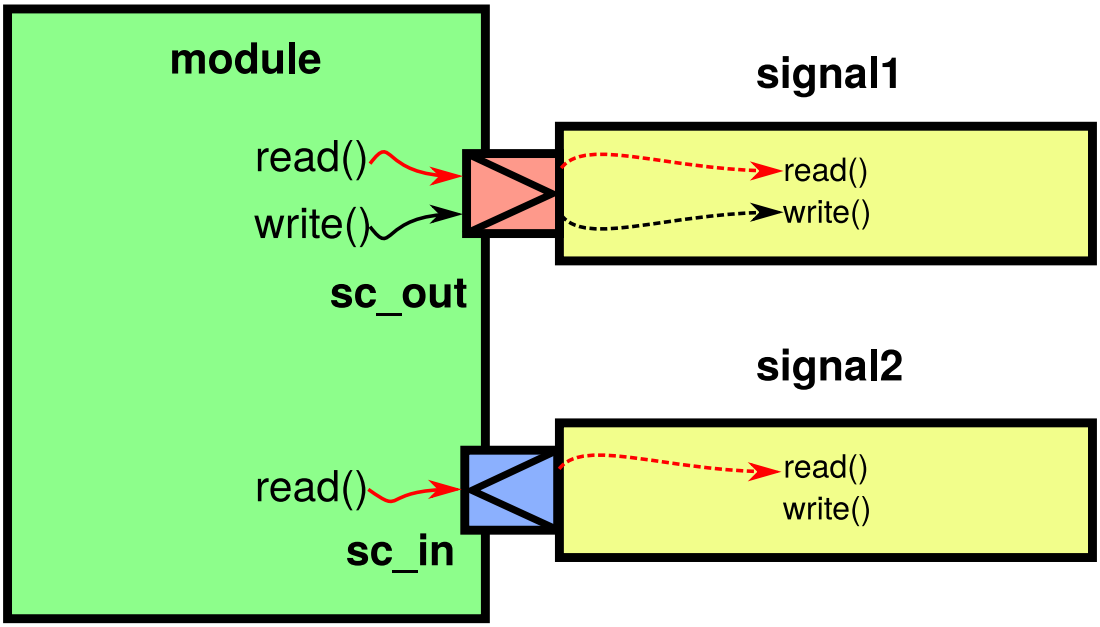

FIG. 4.1: Ports et signaux

```
        « "module : " « name()      « endl
        « "clk    : " « clk.name()  « endl
        « "rst    : " « rst.name()  « endl
        « "data   : " « data.name() « endl
        ;
   }
};

int sc_main(int argc, char * argv[])
{
   foo bar("foobar");
   return 0;
}
```

Voir aussi la macro `SC_NAMED` introduite avec `SystemC 2.3.3`.

Comment utiliser les ports

Les ports SystemC doivent être connectés à des signaux avant de pouvoir les utiliser. Lire ou écrire dans un port non connecté entraine une erreur durant la simulation.

En fonction du type de ports, ils implémentent les méthodes :

- `read()`
- `write()`

Les opérateurs d'affectation sont surchargés pour que l'appel à ces méthodes soit transparent.

Ces méthodes ne font rien à part appeler les méthodes équivalentes des signaux connectés aux ports. Ce sont en réalité des coquilles vide. Quand on lit ou qu'on écrit dans un port, c'est en réalité au signal qui lui est connecté qu'on accède.

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

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

Types résolus

Comme pour les signaux, il existe des ports pour les types résolus :

- `sc_in_resolved`, `sc_out_resolved` et `sc_inout_resolved`
- `sc_in_rv<N>`, `sc_out_rv<N>` et `sc_inout_rv<N>`

Ils doivent être connectés à des signaux du bon type.

Instances

Dans le `sc_main`

Les modules peuvent être instanciés dans la fonction `sc_main`.

Ils peuvent être connectés à des signaux déclarés au même niveau.

L'opérateur `()` est surchargé pour connecter les ports d'un module à des signaux. En interne, la méthode `bind` est appelée et des vérifications sont faites avant le début de la simulation.

Exemple :

```
#include <systemc.h>

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

```
};

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

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

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

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

   return 0;
}
```

Dans un module

Les sous modules sont déclarés comme des attributs du module.

Les connexions se font alors dans le constructeur du module.

Un port peut être connecté soit à :

- un autre port
- un signal interne

Exemple :

```
#include <systemc.h>

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

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

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

   SC_CTOR(foobar)
   {
      // connexions aux I/O
      foo_i.i(i);
      bar_i.o(o);
      // connexion interne
      foo_i.o(s);
      bar_i.i(s);
   }
};

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

   foobar uut("foobar");

   uut.i(i);
   uut.o(o);

   return 0;
}
```

Travail à faire

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

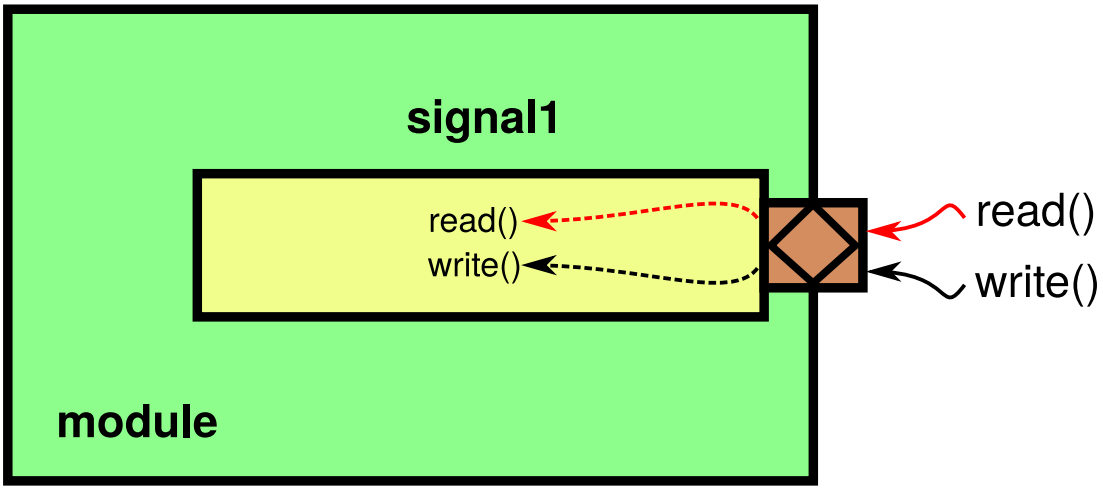

FIG. 4.2: Exports et signaux

Les exports

`sc_export`

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

Sûr et transparent

On peut connecter un `export` directement à un `port`.

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

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

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

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

Exemple :

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

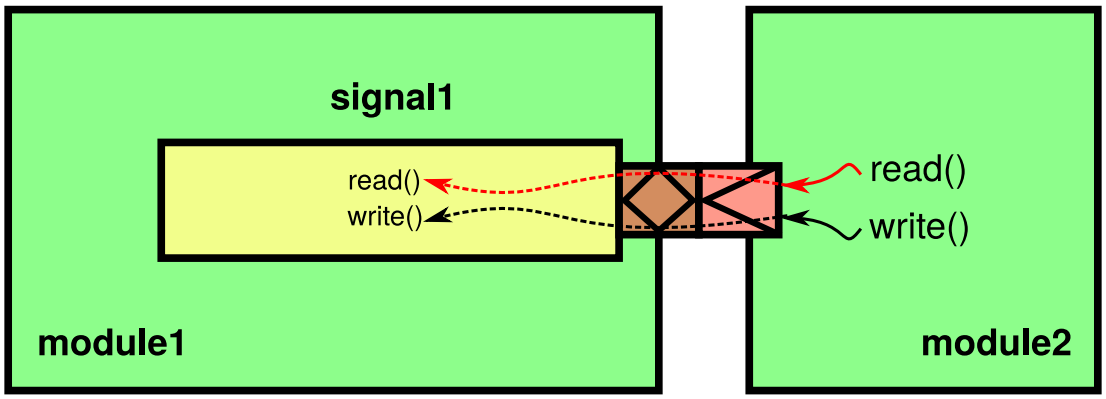

FIG. 4.3: Exports et Ports

```
#include <systemc.h>

SC_MODULE(A)
{
    sc_in<bool> in;

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

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


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

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

```
    }

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

   private:
    sc_signal <bool> sig;

};

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

    a.in(c.inter);

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

    sc_start ();
    cout << "...done" << endl
       <<  "Final: " << c.inter->read()
       << endl;

    return 0;
}
```

---

5 Évènements et processus

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()`

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

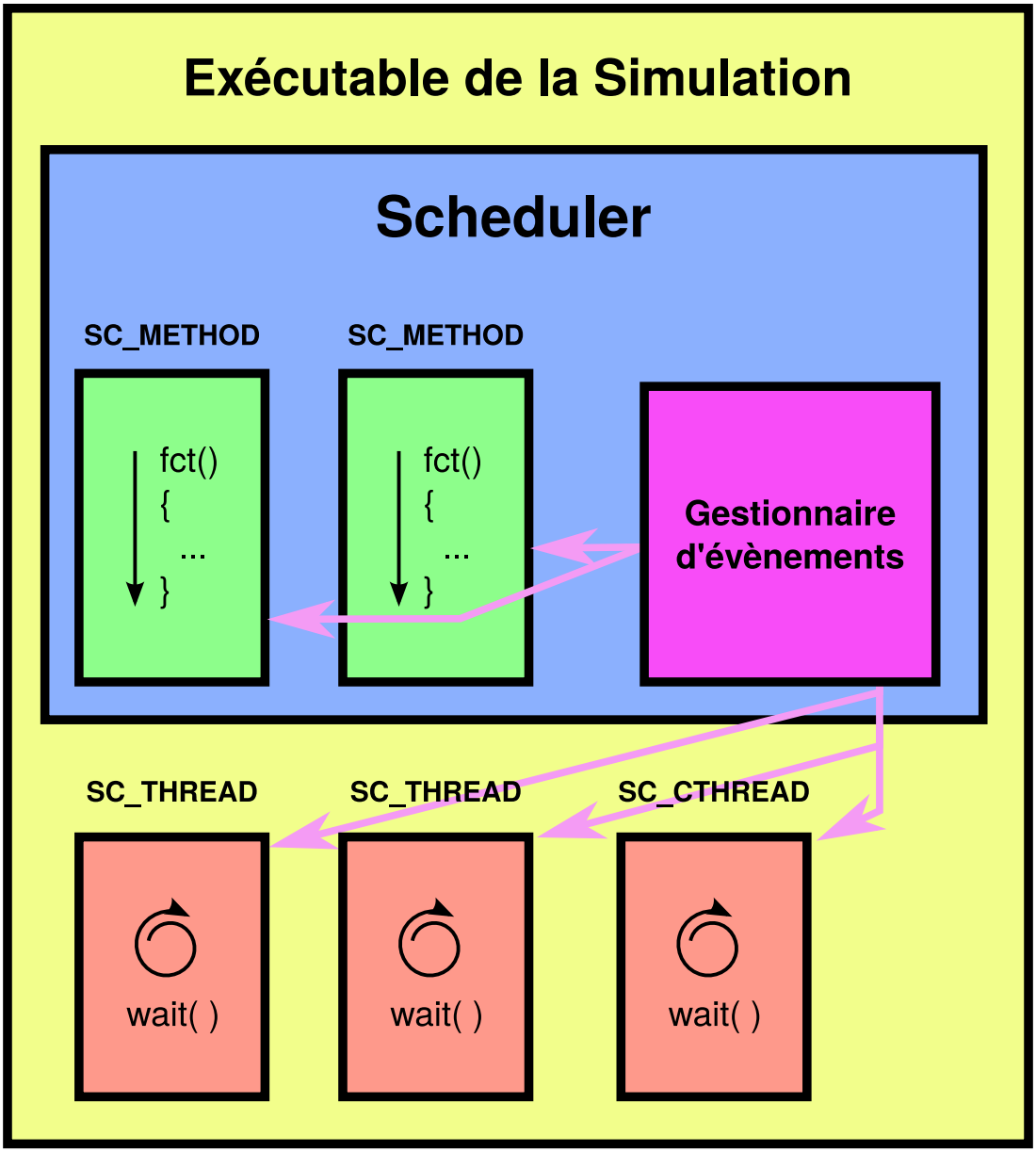

FIG. 5.1: Les processus

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.

6 Niveaux d'abstraction et raffinement

Niveaux d'abstraction et raffinement

Fonctionnel vers RTL

En SystemC on peut partir d'une description fonctionnelle et aller jusqu'à une représentation RTL.

Comment utiliser les processus de SystemC pour faire cela.

Les `SC_METHOD` correspondent aux processus qu'on trouve dans les autres langages HDL. Elles permettent donc de modéliser au niveau RTL.

Pour des modélisations plus haut niveau, les `SC_THREAD` doivent être utilisés. Ils permettent de passer d'une modélisation fonctionnelle à une modélisation précise au bit et cycle près (CABA). Les `SC_THREAD` permettent de faire ces modifications avec le minimum de modification.

Étapes de raffinement

1. Décrire l'algorithme
2. Écrire une version fonctionnelle
3. Définir l'interface du module
   - les entrées/sortie
   - un éventuel protocole
4. Encapsuler la fonction dans un `SC_THREAD`
   - la version fonctionnelle servant de référence
5. Ajouter une information sur le temps en ajoutant des `wait`
   - en nombre de cycles ou en temps absolu
6. Refaire une version RTL… en utilisant des `SC_METHOD`
   - la version précédente servant de référence

Une fonction logicielle prend des arguments et renvoie un résultat. Un module matériel a des entrées et des sorties. En plus, un protocole particulier peut être utilisé pour indiquer que les entrées/sorties sont prêtes.

```
int f(int i, int j ...)
{
```

```
    return xx;
}
```

La première chose à faire est donc de définir un `sc_module` dont les entrées/sortie correspondent au protocole qui sera utilisé. En suite, la fonction peut être encapsulée dans un `SC_THREAD`.

```
SC_MODULE(tt) {
    sc_in<int> i;
    sc_in<int> j;
    ...
    sc_out<int> xx;

    void mthread()
    {
    // Attendre que les entrées soient prêtes
    ...
    // Exécuter la fonction
    xx = f(i,j,..)
    // prévenir que la sortie est prête
    }

    SC_CTOR()
    {
        SC_THREAD(mthread);
        ...
    }
}
```

Dans ce `SC_THREAD` on attend que les entrées soient prêtes, puis on appelle juste la fonction que nous voulons modéliser. Ceci permet d'avoir un premier modèle fonctionnel et respectant le protocole prévu. Par contre, aucune notion de temps n'existe pour l'instant.

Dans un vrai module matériel, l'exécution de la fonction prend du temps (nombre de cycles dans une implémentation séquentielle, par exemple). Si on veut ajouter une notion de temps à notre modèle, on peut, dans un `SC_THREAD` appeler la méthode `wait()` entre l'appel à de la fonction et le renvoie du résultat. Le `SC_THREAD` se mettra alors en veille et le résultat ne sera disponible qu'au bout du temps précisé.

Par exemple

```
SC_MODULE(tt) {
    sc_in<int> i;
    sc_in<int> j;
    ...
    sc_out<int> xx;

    void mthread()
    {
```

```
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   wait(xx,SC_NS);
   // prévenir que la sortie est prête
   }

   SC_CTOR()
   {
      SC_THREAD(mthread);
      ...
   }
}
```

Pour la logique séquentielle, les temps de mise en veille du processus doivent être des multiples de la période d'horloge. De plus, pour garantir le synchronisme, ils doivent dépendre de l'activité d'un signal d'horloge. Dans ce cas, les SC_CTHREAD peuvent simplifier certaines écritures.

Par exemple

```
SC_MODULE(tt) {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   // Latence en nombre de cycles
   static const unsigned int LAT = xxx;

   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   for (int i=0, i<LAT, i++)
      wait();
   // prévenir que la sortie est prête
   }
```

```
   SC_CTOR()
   {
      SC_THREAD(mthread);
      sensitive « clk.pos();
      ...
   }
}
```

Par exemple avec un SC_CTHREAD

```
SC_MODULE(tt) {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   // Latence en nombre de cycles
   static const unsigned int LAT = xxx;

   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // Ajouter la latence
   wait(LAT);
   // prévenir que la sortie est prête
   }

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

Pour augmenter la précision du modèle, il faut pouvoir tracer temporellement l'état des variables internes de la fonction modélisée. Pour cela, il faut pouvoir insérer des attentes (`wait`) dans le corps de la fonction elle-même. Il suffit pour cela que la fonction soit elle-même une méthode du `sc_module`

Par exemple

```
SC_MODULE(tt) {
   // L'horloge
   sc_in<bool> clk;
   // Les autres I/O
   sc_in<int> i;
   sc_in<int> j;
   ...
   sc_out<int> xx;

   int f(int i, intj, ...)
   {
      // étape 1
      ....
      wait(x);
      // étape 2
      ....
      wait(y);
      // ...
      // étape n
      ....
      wait(z);
      return xx;
   }
   void mthread()
   {
   // Attendre que les entrées soient prêtes
   ...
   // Exécuter la fonction
   xx = f(i,j,..)
   // prévenir que la sortie est prête
   }

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

Exemple pratique, Calculer le PGCD

Choix de l'algorithme

Nous voulons modéliser un module matériel calculant le PGCD de deux nombres entiers.

L'algorithme d'Euclide peut être utilisé pour cela. Il existe deux variantes de cet algorithme utilisant :

- des divisions successives ou
- des soustractions successives.

Pour une implémentation matérielle, nous utiliserons la variante avec des soustractions car elle utilisera moins de ressources.

Travail à faire :

Écrire une fonction qui prend deux entiers non signés et qui renvoie leur PGCD en utilisant la variante en soustraction successive de l'algorithme d'Euclide.

Définir les interfaces

Comme l'algorithme est itératif, nous allons l'implémenter en utilisant de la logique séquentielle.

Le module aura l'interface représentée dans le schéma suivant :

- Les données sont des données non signées représentées sur 8 bits.
- Des signaux de contrôle sont ajoutés :
  - le signal `valid` passe à 1, durant 1 cycle, quand les entrées sont prêtes,
  - le signal `ready` passe à 1, durant 1 cycle, quand le résultat est prêt,
  - si `valid` passe à 1 durant le calcul, le comportement est indéterminé.
  - le résultat n'a pas à être maintenu au-delà du cycle durant lequel `ready` est à 1

Travail à faire

1. Écrire un module ayant cette interface.
2. Implémentez le comportement du module en respectant le protocole (`valid`/`ready`).
   - Utilisez pour cela des `SC_CTHREAD`.
3. Instanciez le module dans un `sc_main` où vous testerez exhaustivement le calcul.
   - Vous comparerez le résultat renvoyé par le module au résultat de l'exécution de la fonction de la première étape.
4. Générez les traces des différents signaux.

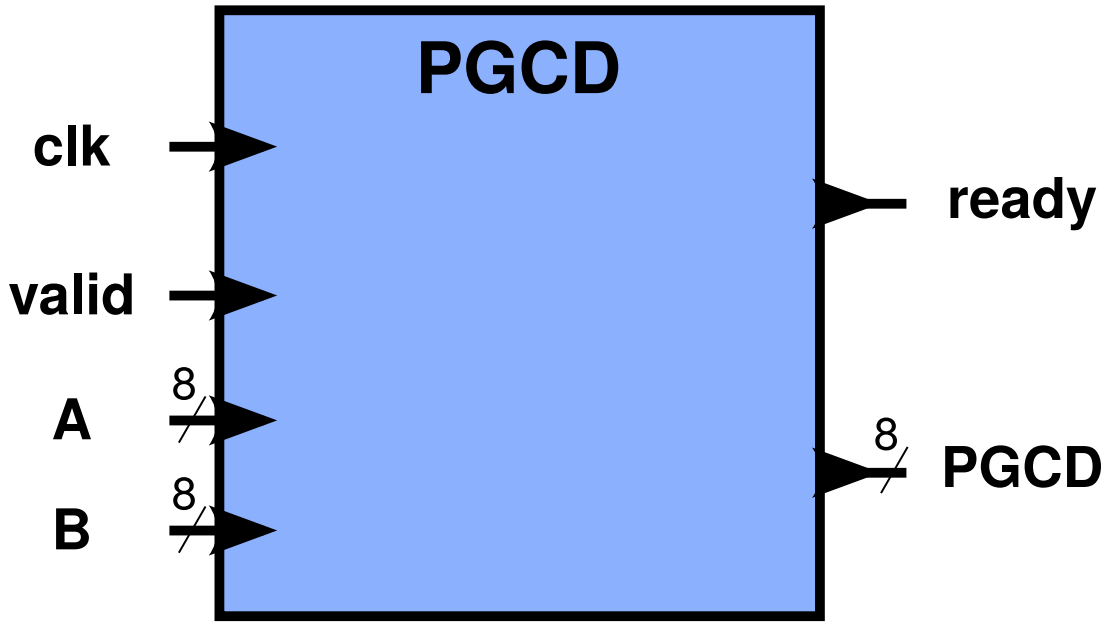

FIG. 6.1: Interface du module PGCD

Modèle temporel

Modifiez le modèle pour avoir un comportement temporel réaliste.

Le nombre d'étapes de calcul pour calculer le PGCD en utilisant l'algorithme d'Euclide dépend des données en entrée.

Dans une implémentation séquentielle simple, chaque étape de calcul peut se faire en un cycle d'horloge.

Travail à faire

1. Modifiez le modèle précédent pour que le comportement temporel soit plus réaliste.
   - Nous voulons obtenir un modèle CABA.
2. Le test précédent devrait fonctionner sans modification.
3. Générez les traces.

Modèle RTL

Cette figure représente le chemin de donnée d'une implémentation possible de cet algorithme.

Travail à faire

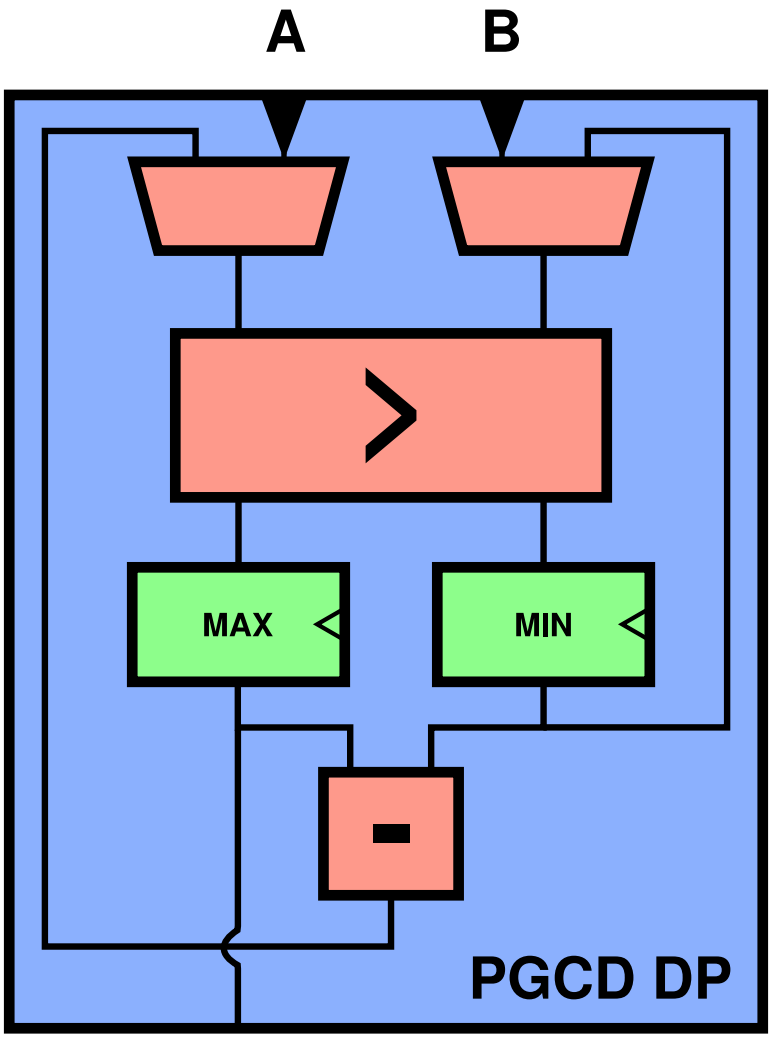

FIG. 6.2: Datapath du PGCD

1. En utilisant exclusivement des `SC_METHOD` proposez une implémentation RTL du module calculant le PGCD.
    - Séparez le chemin de donnée du contrôle
2. Ce module doit avoir une interface compatible avec les versions précédentes.
    - Vous devriez pouvoir tester en parallèle la CABA
    - Observez-vous de différences temporelles ? Expliquez-les.
3. Générez les traces.

---

7 La Co-simulation

Utiliser SyetemC pour la simulation et la vérification de modèle RTL écrits dans d'autres langages.

Les modèles RTL sont généralement écrits en utilisant les sous-ensembles synthétisables des langages HDL communs (VHDL, Verilog/SystemVerilog)

Même dans ce cas, SystemC peut être utilisé pour concevoir un environnement de simulation pour vérifier et valider le modèle RTL.

Les avantages de l'utilisation de SystemC viennent principalement de :

- C++ permet de structurer/réutiliser l'environnement,
- Langage compilé *efficace*,
- utiliser les bibliothèques systèmes,
- interaction poussée avec le simulateur,
- se passer d'outils propriétaires.

D'un autre côté :

- C++ est un langage complexe pas forcément maitrisé par les concepteurs HDL,
- il existe des alternatives (*DPI/VPI*) pour utiliser des bibliothèques système dans un environnement de simulation en SystemVerilog,
- certaines méthodologies de vérification font référence dans l'industrie et elle n'utilisent pas forcement SyetemC.

Mixer les langages

La co-simulation

La **co-simulation** consiste en la mise en œuvre d'une simulation dans laquelle les éléments ont été écrits dans des langages différents.

Pour pouvoir le faire :

- les concepts manipulés doivent pouvoir s'exprimer dans les deux langages,
- l'outil (*simulateur*) doit les supporter et permettre de passer de l'un à l'autre.

Si on doit faire une simulation évènementielle, les langages utilisés doivent prévoir les concepts :

- d'évènements,
- de processus.

De plus, on doit pouvoir représenter la structure des *modules* simulés dans les deux langages.

Les langages RTL communément utilisés (VHDL, Verilog/SystemVerilog) ont été conçus autour de ces concepts. SystemC ajoute ces concepts à C++.

De plus, il faut pouvoir exprimer les données manipulées avec la bonne granularité (entier, bits, vecteurs/bus…) dans les deux langages. Il n'est généralement pas possible de faire passer d'un langage vers l'autre des structures de données plus complexes (interfaces SystemVerilog, des structures…).

Généralement le point bloquant vient de l'outil utilisé :

- que supporte-t-il dans le langage A et B,
- que sait-il transmettre de A vers B,
- comment l'indiquer à l'outil ?

Testbench en SystemC

Ce que nous allons faire ici c'est simuler un design HDL décrit en Verilog/SystemVerilog dans un environnement en SystemC.

Cet environnement contiendra de quoi générer des stimuli et vérifier les réponses du bloc HDL (*DUT :* Design Under Test).

Il serait aussi envisageable d'insérer un module SystemC dans un banc de test (*testbench*) écrit dans un autre langage, mais ce ne sera pas présenté dans ce cours.

Mise en pratique

Deux approches

Nous allons voir deux exemples de simulation d'un module RTL (Verilog/SystemVerilog) dans un testbench écrit en SyetemC.

Deux approches :

- Utiliser un simulateur qui accepte les deux langages (**Modelsim**).
- Utiliser un convertisseur (**Verilator**) qui transforme le RTL dans un langage que l'on peut compiler.

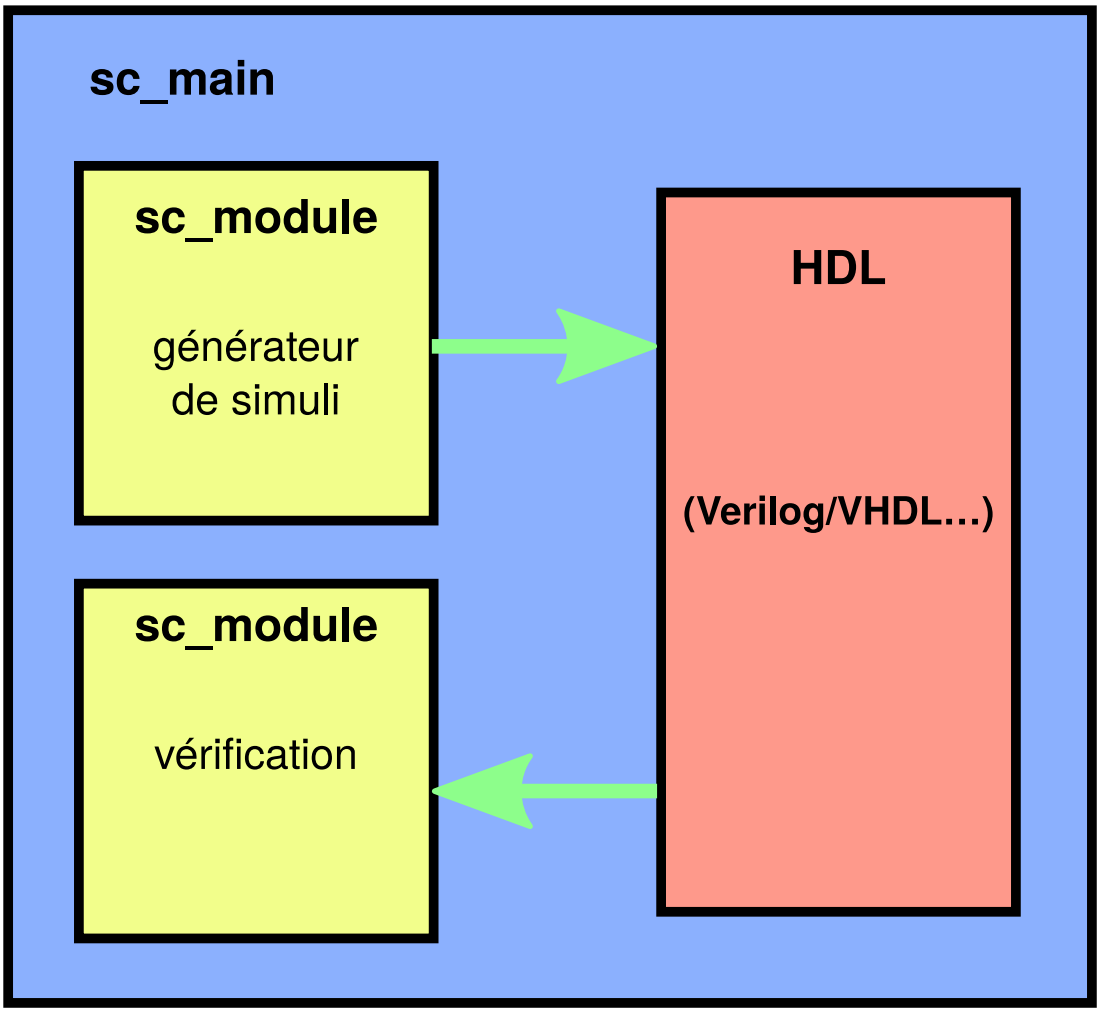

FIG. 7.1: Co-simulation

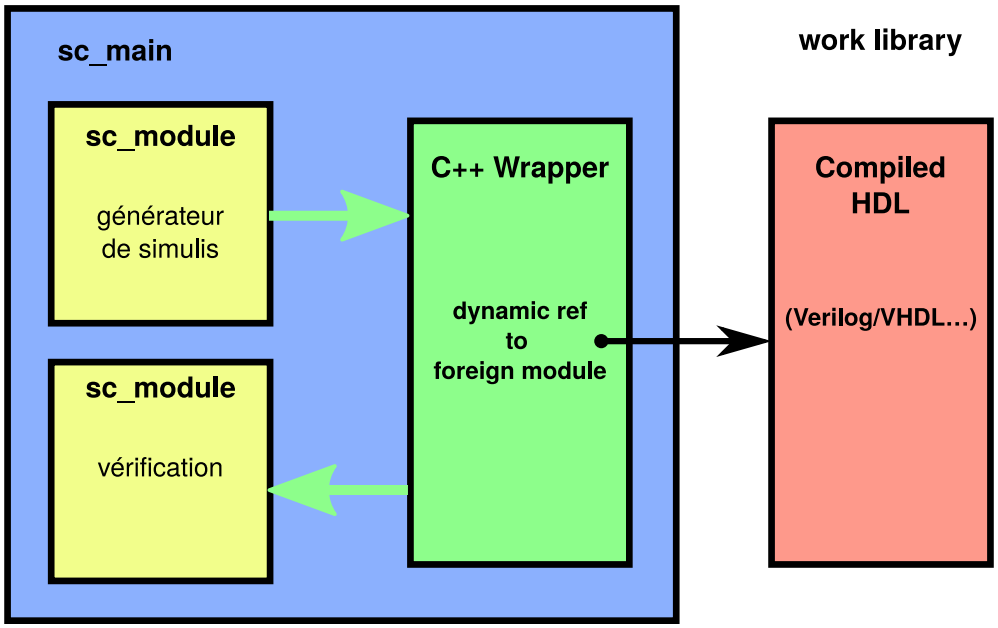

FIG. 7.2: Foreign Module

Ici nous allons mettre en œuvre deux approches differentes :

1. En utilisant Modelsim, un simulateur commercial qui supporte l'utilisation de plusieurs langages HDL, nous pouvons compiler séparément les différents fichiers sources. La seule contrainte viendra du fait que le code SystemC étant du C++ il faudra d'une que les modules HDL soient définis d'une scertaine façon au moment de la compilation des fichiers C++ qui y font référence. Le simulateur s'occupera par la suite de faire communiquer l'ensemble de façon transparente.
2. En utilisant Verilator un outil qui permet de générer des modèles C++ et SystemC à partir de modèles en Verilog/SystemVerilog. C'est un outil libre (OpenSource) destiné à générer des modèles de simulation à partir de modèles RTL synthétisables. Le tout est ensuite compilé et le simulateur intégré à la bibliothèque SystemC peut alors être utilisé.

Modelsim

Module "étranger"

Avec Modelsim on doit utiliser un module *wrapper* SystemC qui est compilé avec le testbench. Ce module est une coquille vide qui ne contient pas de code comportemental, mais doit :

- reprendre l'interface du module HDL (les entrées/sorties) pour permettre de faire les connexions,
- utiliser un mécanisme (`foreign module`) du simulateur pour le lier au module HDL pour mettre en relation le wrapper avec le module HDL durant la phase d'élaboration.

Ce wrapper peut être écrit à la main ou généré automatiquement après compilation du module HDL (`scgenmod --help` pour plus d'informations.)

Exemple (HDL) :

```
module counter ( input clk, nrst, enable,
                 output logic[7:0] Q
               );

// ...

endmodule
```

Exemple (Wrapper) :

```
#ifndef _SCGENMOD_counter_
#define _SCGENMOD_counter_

#include "systemc.h"

class counter : public sc_foreign_module
{
public:
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;


    counter(sc_module_name nm, const char* hdl_name)
     : sc_foreign_module(nm),
       clk("clk"),
       nrst("nrst"),
       enable("enable"),
       Q("Q")
    {
        elaborate_foreign_module(hdl_name);
    }
```

```
    ~counter()
    {}

};

#endif
```

Contrairement à un `sc_module`, ici on hérite d'un `sc_foreign_module` qui est une classe propre à Modelsim.

L'interface de ce module (ses entrées/sorties) correspondent à celles du module HDL. Les types qui existent dans les deux langages ne sont pas les même, Modelsim (se référer à la documentation de l'outil) prévoit plusieurs possibilités. Dans cet exemple, on fait correspondre aux bits Verilog des `bool` et aux vecteurs des `sc_uint<>` mais nous aurions pu utiliser respectivement des `sc_logic` et des `sc_lv`.

Le constructeur de ce module prend 2 arguments, l'un correspondant au nom de l'instance (`sc_module_name`) au sens SystemC et l'autre correspondant au nom du module HDL auquel il doit être lié. Sinon, le constructeur ne fait qu'appeler une méthode `elaborate_foreign_module` qui déclenchera un mécanisme interne du simulateur pour l'élaboration du module RTL.

Exemple (Instanciation) :

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

int sc_main(int argc, char * argv[]) {
   //..
   //                inst. name   RTL module
   counter counter_0("counter_0", "counter");
   counter_0.clk( xx );
   //...
   counter counter_1("counter_1", "counter");
   counter_1.clk( xx );
   //...
   return 0;
}
```

Ensuite, le module wrapper peut être instancier et utilisé comme un module SystemC standard.

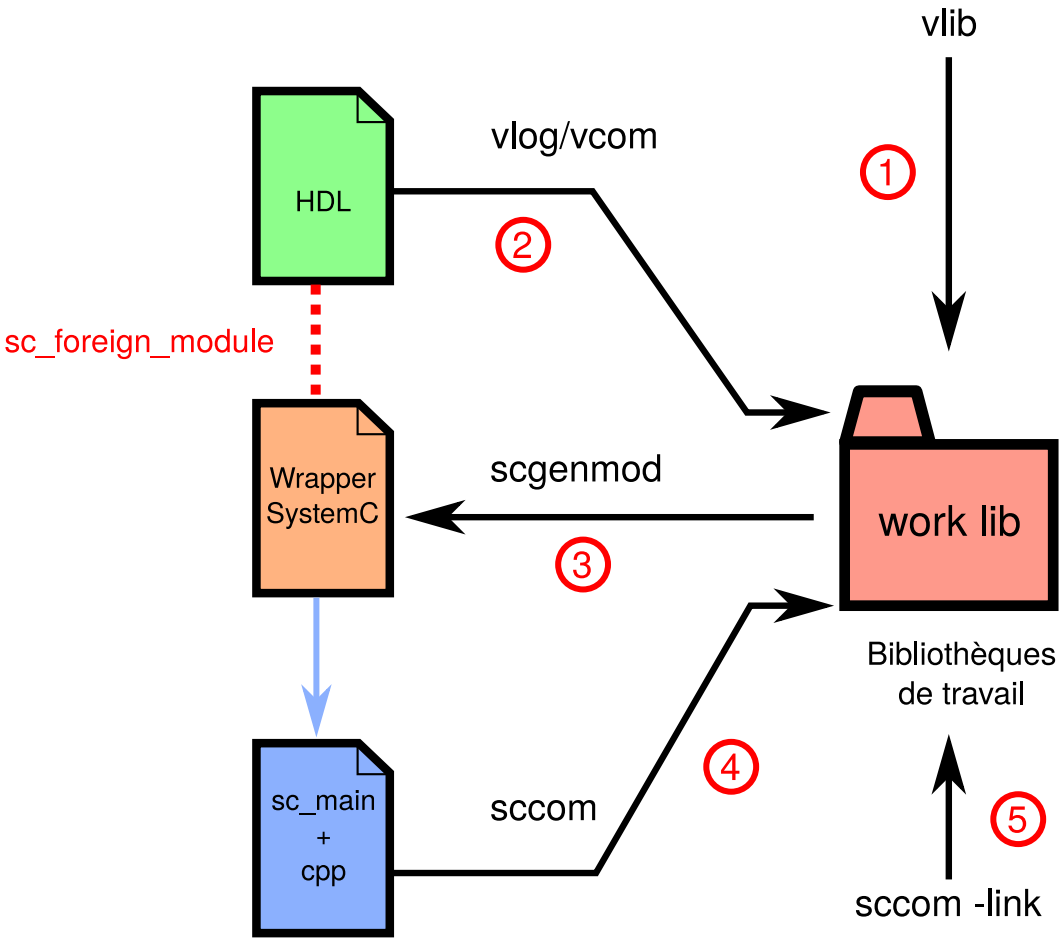

FIG. 7.3: Foreign Module

Workflow `Modelsim`

La génération du module wrapper peut être automatisée en utilisant la commande `scgenmod` de Modelsim après compilation du module HDL. `scgenmod` possède aussi des options pour préciser les correspondances entre les types HDL et SystemC.

La commande de Modelsim pour compiler des fichiers SystemC est `sccom`. Elle utilise en interne une version de g++ fournie avec l'outil. Aussi, `sccom --link` doit être appelé après la compilation pour l'édition de liens.

`sccom` définit la macro `MTI_SYSTEMC` qui peut être testée pour écrire du code spécifique pour Modelsim.

`Verilator`

Convertir le Verilog en C++/SystemC

Utiliser un “convertisseur” qui converti le Verilog en C++/SystemC :

- `Verilator` : https ://www.veripool.org/wiki/verilator

Verilator est un outil qui analyse du code HDL (seulement Verilog ou SystemVerilog) et qui génère un AST (*Abstract Syntaxic Tree*).

À partir de cet AST il peut :

- générer un modèle SystemC ou C++ (qui utilise son propre modèle de simulation),
- mettre en œuvre un certain nombre d’optimisations :
  - simplification de logique combinatoire,
  - propagation de constantes,
  - faire les calculs sur des tailles adaptées aux processeurs…

Il a été développé pour générer des simulateurs rapides à partir de modèles RTL.

C’est un outil libre dont le code source est distribué sous les termes de la licence LGPLv3. Le code source est disponible ici et des versions binaires sont disponibles dans la majorité des distributions Linux.

Conversion automatique du RTL grâce à Verilator

Verilator permet donc de générer un modèle de simulation SystemC à partir d’un module Verilog ou SystemVerilog.

Le module généré peut alors être intégré dans une simulation SystemC standard.

Workflow `Verilator`

---

1. On utilise Verilator pour générer des modèles SystemC à partir des modules Verilog/SystemVerilog.
2. L’ensemble des sources est compilé en utilisant le compilateur C++ natif.
3. En plus de la bibliothèque SystemC, l’édition de liens doit utiliser certaines bibliothèques fournies avec l’outil.

L’outil sait aussi générer un *Makefile* pour automatiser la compilation.

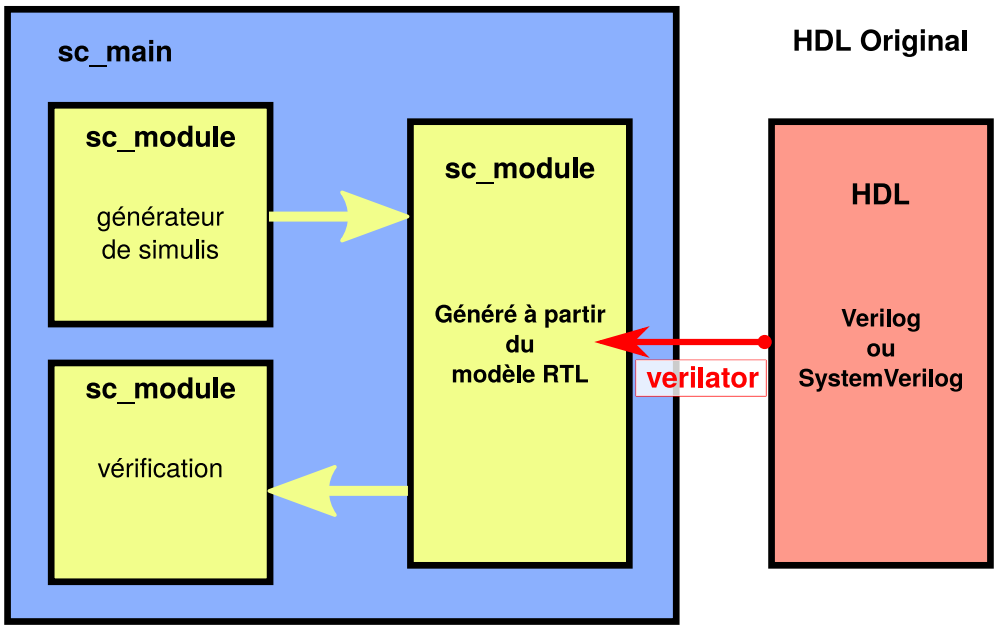

FIG. 7.4: Convertir le Verilog en SystemC

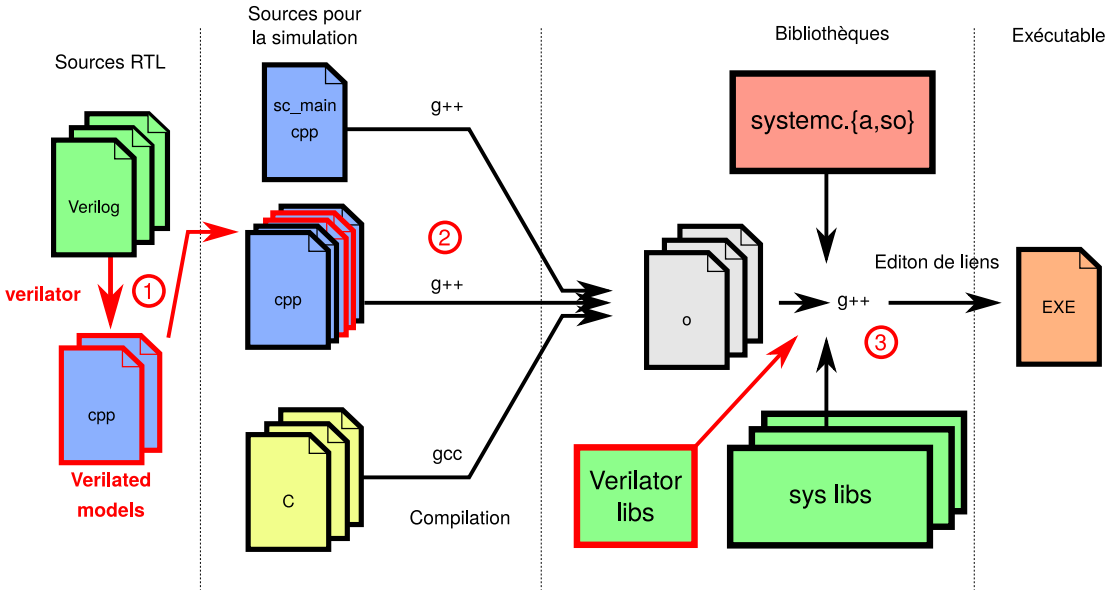

FIG. 7.5: Workflow

Exemple (Conversion) :

Pour le module Verilog précédent (`counter`) on obtient le module SystemC `Vcounter` :

```
#ifndef _Vcounter_H_
#define _Vcounter_H_

#include "systemc.h"
#include "verilated_sc.h"
#include "verilated.h"

SC_MODULE(Vcounter) {
  public:
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;

    ....
}

#endif
```

Ce modèle peut être instancié directement dans un autre module SystemC ou directement dans le `sc_main`.

Pour obtenir ce module `verilator` est utilisé avec les options suivantes :

```
verilator -sc --pins-sc-uint counter.sv
```

La première option indique que l'on veut un modèle SystemC, la seconde qu'on veut des `sc_uint` pour les signaux de l'interface dont la largeur est supérieure à 1.

Si l'on souhaite spécialiser le code, la macro `VM_SC` peut être testée.

En réalité, le module généré est un peu plus complexe et contient des références à des objets internes de l'outil.

```
// Verilated -*- SystemC -*-
// DESCRIPTION: Verilator output: Primary design header
//
// This header should be included by all source files instantiating the design.
// The class here is then constructed to instantiate the design.
// See the Verilator manual for examples.

#ifndef _Vcounter_H_
#define _Vcounter_H_
```

```
#include "systemc.h"
#include "verilated_sc.h"
#include "verilated.h"

class Vcounter__Syms;

//----------

SC_MODULE(Vcounter) {
  public:

    // PORTS
    // The application code writes and reads these signals to
    // propagate new values into/out from the Verilated model.
    sc_in<bool> clk;
    sc_in<bool> nrst;
    sc_in<bool> enable;
    sc_out<sc_uint<8> > Q;

    // LOCAL SIGNALS
    // Internals; generally not touched by application code

    // LOCAL VARIABLES
    // Internals; generally not touched by application code
    VL_SIG8(__Vcellout__counter__Q,7,0);

    // INTERNAL VARIABLES
    // Internals; generally not touched by application code
    Vcounter__Syms* __VlSymsp;  // Symbol table

    // PARAMETERS
    // Parameters marked /*verilator public*/ for use by application code

    // CONSTRUCTORS
  private:
    VL_UNCOPYABLE(Vcounter);  ///< Copying not allowed
  public:
    SC_CTOR(Vcounter);
    virtual ~Vcounter();

    // API METHODS
  private:
    void eval();
```

```
  public:
    void final();

    // INTERNAL METHODS
  private:
    static void _eval_initial_loop(Vcounter__Syms* __restrict vlSymsp);
  public:
    void __Vconfigure(Vcounter__Syms* symsp, bool first);
  private:
    static QData _change_request(Vcounter__Syms* __restrict vlSymsp);
    void _ctor_var_reset();
  public:
    static void _eval(Vcounter__Syms* __restrict vlSymsp);
  private:
#ifdef VL_DEBUG
    void _eval_debug_assertions();
#endif // VL_DEBUG
  public:
    static void _eval_initial(Vcounter__Syms* __restrict vlSymsp);
    static void _eval_settle(Vcounter__Syms* __restrict vlSymsp);
    static void _settle__TOP__1(Vcounter__Syms* __restrict vlSymsp);
} VL_ATTR_ALIGNED(128);

#endif // guard
```

Par défaut, ce fichier d'en-tête, ainsi qu'un fichier C++, sont générés dans un sous-répertoire `obj_dir`. La méthode `eval` est l'unique méthode SystemC (`SC_METHOD`) dans laquelle se trouve tout le comportement du module RTL. L'observation des signaux interne n'est pas donc possible par défaut pour obtenir la meilleure performance.

On peut aussi demander à Verilator de générer un `Makefile` pour automatiser la compilation. Par exemple :

```
verilator -sc --pins-sc-uint -exe counter.sv sc_main.cpp -o ../sim.exe
```

Générera un Makefile (`Vcount.mk`) dans le répertoire `obj_dir` permettant de compiler un exécutable nommé `sim.exe` dans le répertoire courant.

Pour obtenir la documentation de l'outil et la liste des options :

```
verilator --help
```

Travail à faire :

Reprendre l'exemple du PGCD.

Voici le code SystemVerilog d'un module RTL calculant le PGCD de deux nombres codés sur 8 bits. Il possède la même interface et le même comportement que le module demandé dans le chapitre "Niveaux d'abstraction et raffinement".

```
module PGCD (
               input clk,
               input valid,
               input [7:0] a,
               input [7:0] b,
               output ready,
               output [7:0] pgcd
            );

logic [7:0] max, min;
wire [7:0] d = max-min;

always_ff @(posedge clk)
   if (valid)
   begin
      max <= (a>b)? a:b;
      min <= (a>b)? b:a;
   end
   else
   begin
      max <= (d>min)? d : min;
      min <= (d>min)? min : d;
   end

assign pgcd = max;
assign ready = (min==max);

endmodule
```

Travail à faire : mettre en place un environnement de simulation en SystemC en reprenant le `sc_main` existant, permettant de simuler ce module SystemVerilog en utilisant Modelsim et Verilator.

- Le `sc_main` doit instancier un module `testeur` qui génère des séquences de test et vérifie la validité du résultat renvoyé.
- Le code du `sc_main` doit être compatible avec les deux outils.
- **Bonus** écrire un Makefile qui automatise la compilation dans les deux cas.

8 Les canaux de communication

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;
}
```

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.

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 ?

`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`.

---

9 Mini-projet

Objectifs :

Le but de ce mini-projet est de vous faire modéliser un système de traitement vidéo, qui prend en entrée un flux vidéo et qui lui applique au vol des transformées basiques.

On cherchera dans ce projet à être le plus efficace possible, en termes de ressources pour la simulation, de types de variables/signaux, de complexité et de lisibilité du code.

Plan du projet

Le travail demandé pour ce mini-projet se décompose en trois phases :

- récupération d'un module de génération de flux vidéo et de son environnement de test
- écriture d'un système de réception de flux vidéo
- écriture du système de transformation vidéo, et test de ce système.

Contexte

On cherche à modéliser et simuler un système de traitement vidéo, qui reçoit un flux vidéo en entrée, et qui restitue en sortie un flux transformé au même format. Les transformations appliquées peuvent être multiples :

- filtrage (flou léger)
- zoom in / zoom out
- …

Le format de flux vidéo choisi est une version simplifiée CCIR 601. En voici les grandes lignes (mais le web regorge d'information à ce sujet) :

- Le flux vidéo est transmis sur un bus comprenant 3 parties
  - une horloge à la fréquence pixel
  - les pixels, sous forme d'un quadruplet RGBA (8 bits pour chacune des composantes R, G, et B, et 8 bits pour l'alpha. 0=transparent, 255=opaque).
  - deux signaux de synchronisation : `HREF` et `VREF`.
- Une ligne vidéo se compose **d'environ** 864 pixels, dont **exactement** 720 actifs (c'est-à-dire qui contiennent des pixels valides. Les autres sont en nombre indéterminé, ils correspondent au temps nécessaire pour un retour ligne).

- Une image se compose **d'environ 625 lignes**, dont **exactement** 576 actives (les autres correspondent au temps de retour trame).

Pendant la transmission de pixels actifs d'une ligne, un signal de synchronisation, `HREF`, est mis à 1. Pendant le reste du temps, `HREF` vaut 0.

Au début d'une trame, un signal de synchronisation verticale, `VREF`, est mis à 1 pendant 3 lignes. Le front montant de `VREF` coïncide avec celui de `HREF`.

Les pixels sont transmis à la cadence de `13.5Mhz`, synchrones sur front montant de l'horloge.

Module d'entrée

Pour tester votre système, vous allez avoir besoin de modules d'entrée et de sortie vidéo :

- un module doit permettre de générer un flux vidéo à partir d'une séquence d'images,
- un autre module, doit permettre de générer des images à partir d'un flux pour vérifier visuellement le résultat.

Pour cela, nous vous proposons d'utiliser une série d'images au format PNG. Les fonctions de la bibliothèque `libpng`, une bibliothèque libre, permettront de lire et d'écrire *simplement* des images au format PNG.

On aurait pu utiliser d'autres bibliothèques pour lire directement des flux vidéo, mais cela aurait été un peu plus compliqué.

Le module de génération de flux vidéo vous est fourni. Il est disponible dans le dépôt `Git` suivant :

```
git@gitlab.enst.fr :se207/projet.git
```

Vous pouvez, cloner le dépôt séparément et copier manuellement les fichiers dans votre dépôt personnel ou le fusionner (`merger`) directement avec votre dépôt personnel comme suit :

```
git remote add projet git@gitlab.enst.fr :se207/projet.git
git remote update
git checkout master
git pull projet master --allow-unrelated-histories
```

Vous obtiendrez à la racine du dépôt un dossier `projet_video` contenant :

- un fichier de fonctions pratiques pour accéder simplement aux fonctionnalités de `libpng` (`image.{h,c}`)
- le module de génération de flux vidéo proprement dit (`video_in.{h,cpp}`)
- un environnement de test sommaire (`system.cpp`)
- un `Makefile`
- des images de test, tirées d'un film d'animation bien connu.

Pour des raisons de simplicité on se limite à des images en niveaux de gris. Les pixels venant du générateur de flux ne sont donc codés que sur **8bits**.

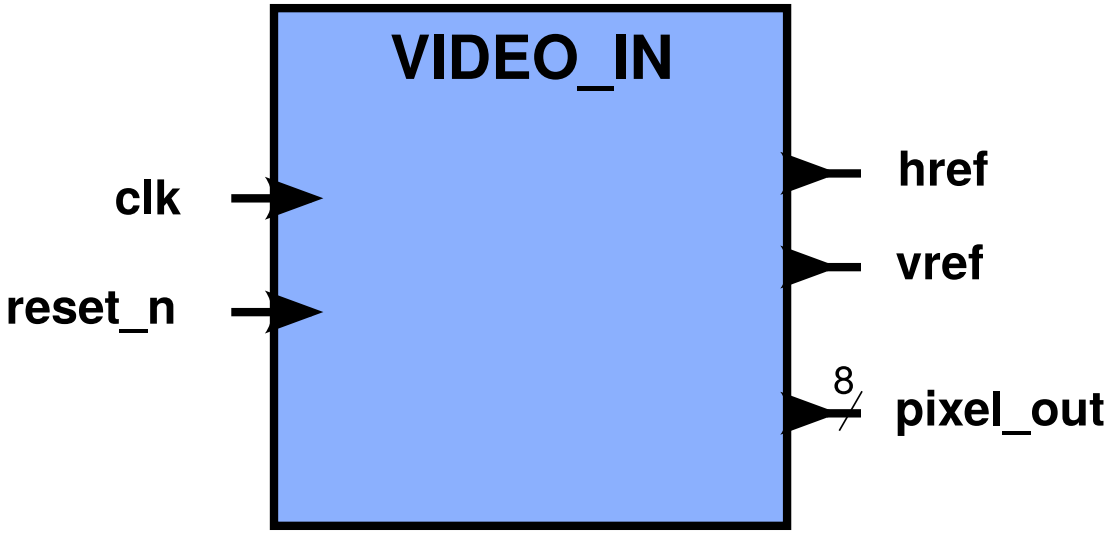

FIG. 9.1: Module générant le flux

Vous pouvez simuler le système tel quel et observer les traces générées (`simulation_trace.vcd`). Prenez le temps d'analyser ces chronogrammes et de vous familiariser avec le format *pseudo-CCIR601*.

Au travail :

module de sortie

Maintenant que vous avez un module qui vous fournit un flux vidéo, il serait sympa d'avoir un module qui fait l'inverse, pour vérifier visuellement les transformations vidéo.

C'est le rôle d'un module que vous allez concevoir et vous appellerez `video_out`.

Le travail à faire :

- écrire un module de récupération du flux vidéo, qui crée des images PNG à partir du flux vidéo.
  - Le nom du fichier généré doit dériver du nom du module une fois instancié.
- instancier dans votre environnement de test les deux modules `video_in` et `video_out`
- relier les deux modules directement entre eux et simuler le tout.

Les images de sorties doivent être les mêmes que celles d'entrée. Vous pouvez le vérifier en les affichant successivement. Par exemple :

```
display iinput0.png output0.png
eog iinput0.png output0.png
```

…

Faites attention, vérifiez bien qu'il n'y a pas de décalage, même d'un pixel, en début et fin d'image.

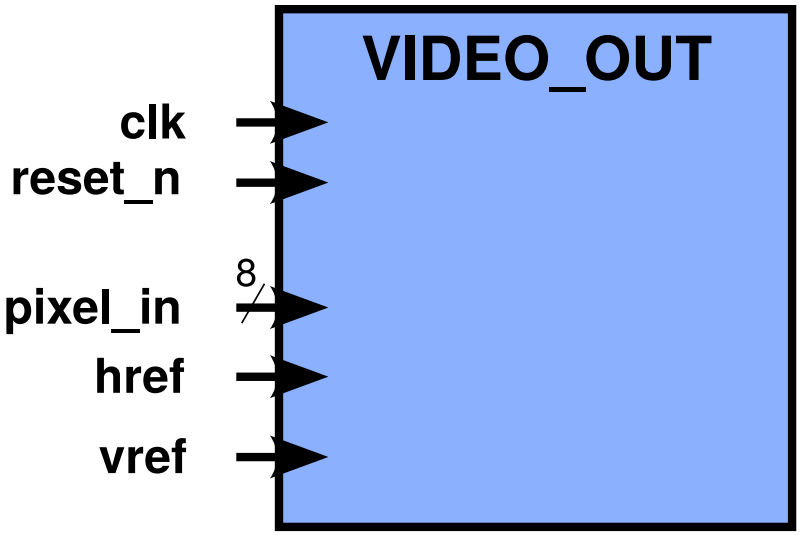

FIG. 9.2: Module récupérant le flux

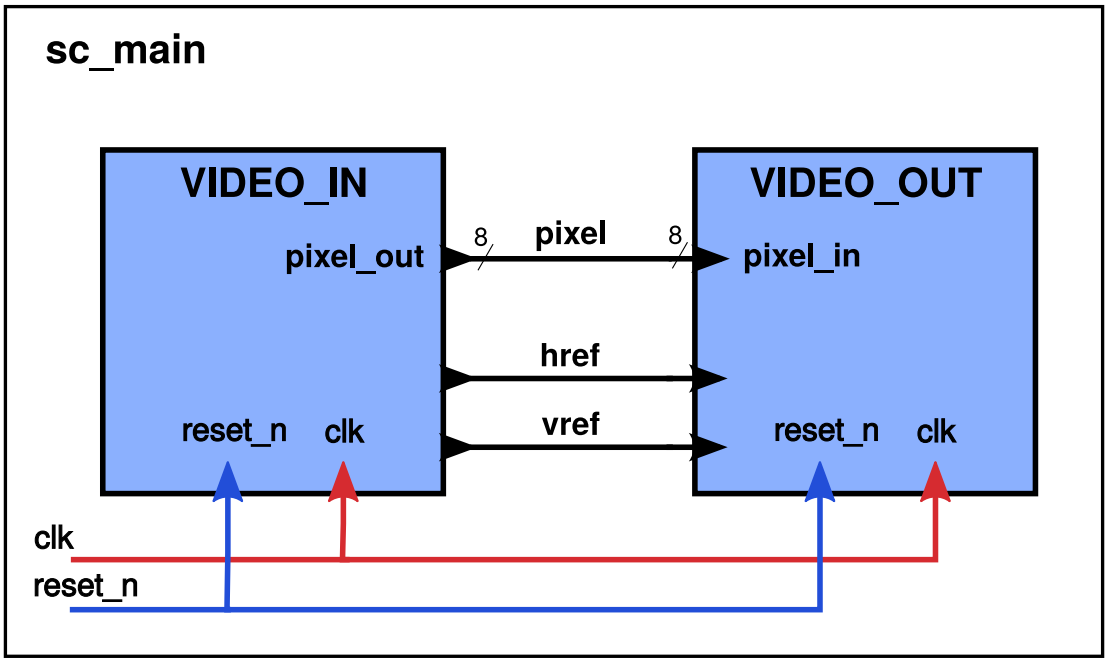

FIG. 9.3: Premier testbench

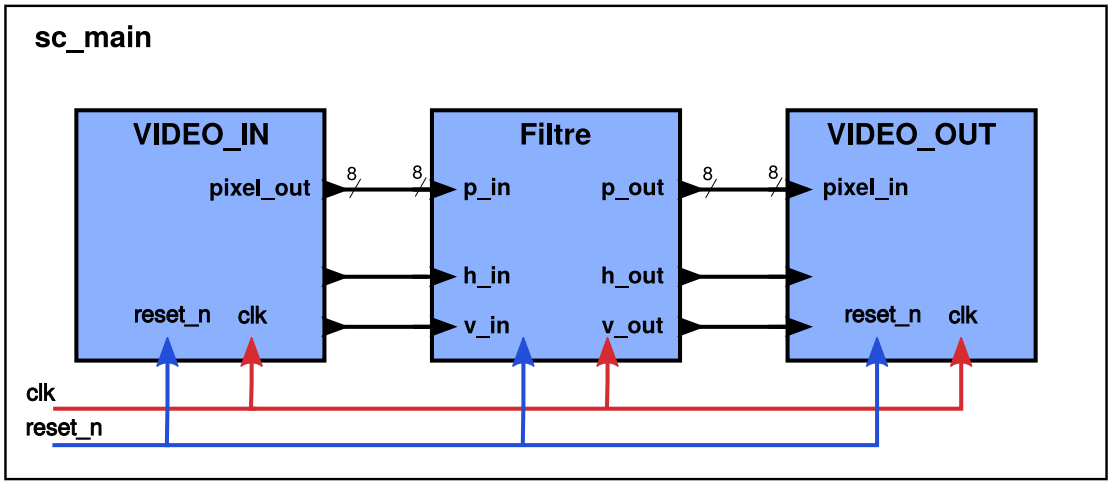

FIG. 9.4: Testbench d'un module de filtrage

Attention :

Votre module `video_out` ne doit pas compter sur le fait que les trames font 625 lignes ni 864 pixels ! C'est à lui de se synchroniser (à l'aide des signaux HREF et VREF) sur le flux entrant. Il peut par contre compter sur le fait que dans chaque trame il y aura **exactement** 720×576 pixels valides.

Effets spéciaux

Le but du jeu est maintenant de construire un système qui va opérer, à la volée, sur un flux sortant du premier module.

Vous avez d'ores-et-déjà un module qui fournit des flux vidéos et un module qui en génère des PNG. Reste à écrire ce qui va s'interposer entre les deux.

On implémentera ces filtres (dans l'ordre) :

- filtre moyenneur sur 8 pixels adjacents (pixel central + ses 8 voisins) : dans la mesure du possible, ce filtre doit effectuer son traitement au vol.
- zoom ×2 : on fera un zoom centré, en dupliquant chaque pixel. Un traitement au vol est-il possible et/ou adapté ?

Vérifiez visuellement que tout marche bien et que les signaux de synchronisation HREF et VREF respectent bien le format initial.

Si tout fonctionne correctement, vous devriez pouvoir chainer les différents modules en les instanciant les uns après les autres.

- **Bonus :** Le modèle du filtre moyenneur pourrait être utilisé pour tous les filtres agissant sur le même type de voisinage 3×3 (Sobel, Gaussien…). Modifiez le filtre moyenneur pour pouvoir préciser les paramètres du filtre à la construction du module.

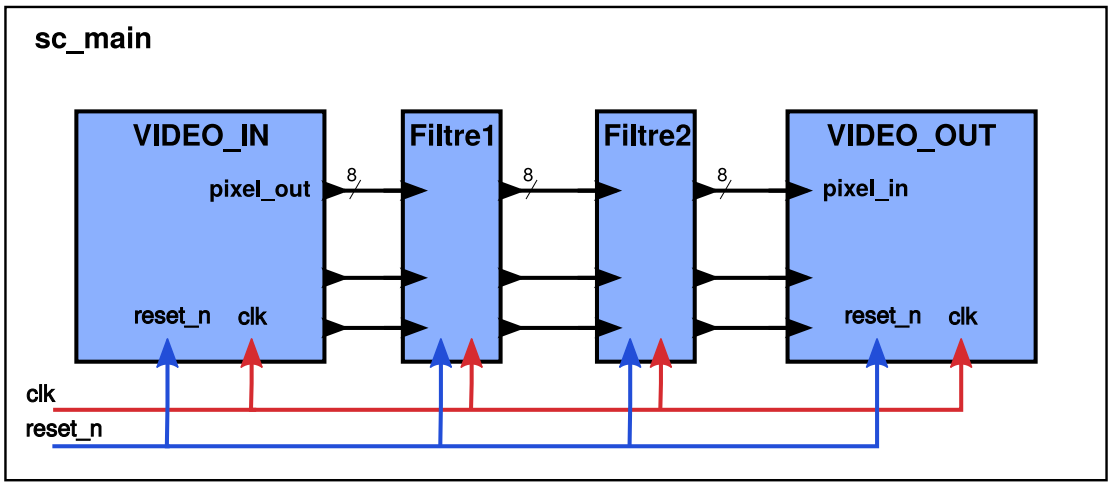

FIG. 9.5: Testbench avec plusieurs modules de filtrage